Estudio y experimentación sobre la programación multinúcleo y extensiones SIMD

Juan Deus Lorenzo, David Landín Calvo

8 de mayo de 2019

1. Introducción

1.1. Objetivos y procedimiento

Al resolver esta práctica se busca demostrar como de eficiente es la optimización O2 del compilador icc respecto de las versiones optimizadas por el programador de forma secuencial, vectorizada y multihilo. Se puede predecir que va a haber situaciones donde unas versiones sean mejores que otras y para ello se explorarán varios casos. Para la vectorización se usarán las extensiones SSE3 y para la programación multihilo la API *OpenMP*.

El procedimiento consistirá en realizar un cómputo complejo para el procesador utilizando multiplicaciones, potencias y sumas de distintas cantidades de cuaterniones aleatorios para medir el número de ciclos necesitado. Se propondrán distintas formas de llevar a cabo dicho cómputo siguiendo las técnicas comentadas en el párrafo anterior y se verá que en función de la cantidad de cuaterniones con los que haya que operar habrá versiones mejores que otras.

La cantidad de cuaterniones del cómputo será 10^q con q = 2, 4, 6 y 7. Para cada versión y para cada uno de los tamaños se harán 10 medidas y se escogerá la mediana como resultado representativo. En el caso de *OpenMP* para cada tamaño se probará con 1, 2, 4, 8 y 16 hilos (cada uno 10 medidas).

Código 1: pseudocódigo del cómputo

```
Entrada:

    a(N), b(N): N cuaterniones de floats aleatorios.

Salida:

    dp: cuaternion que recoge el sumatorio.

Algoritmo:

    c(N): vector auxiliar de quaterniones

    for i = 1, N
        c(i) = a(i)*b(i);

    dp = 0;

    for i = 1, N
        dp = dp + c(i)*c(i)
```

1.2. Características del procesador

La experimentación fue llevada a cabo en un nodo integrado del SVG del *Finis Terrae II* del CESGA a través de una cola *shared*. Este nodo tiene 2 procesadores *Intel Xeon E5-2650 v3* de microarquitectura *Haswell* de 22 nms con una frecuencia base de 2,3 GHz (hasta 3,0 GHz turbo) y 10 núcleos físicos con 2 hilos cada uno (en total 20 hilos).

La memoria caché de este procesador está organizada de la siguiente forma:

1. Caché de primer nivel de 32 KB con 8 vías de asociatividad.
   a) Caché de datos individual para cada núcleo físico.
   b) Caché de instrucciones individual para cada núcleo físico.
2. Caché de segundo nivel de 256 KB con 8 vías de asociatividad unificada individual para cada núcleo físico.
3. Caché de tercer nivel de 25 MB con 20 vías de asociatividad compartida por los 10 núcleos físicos.

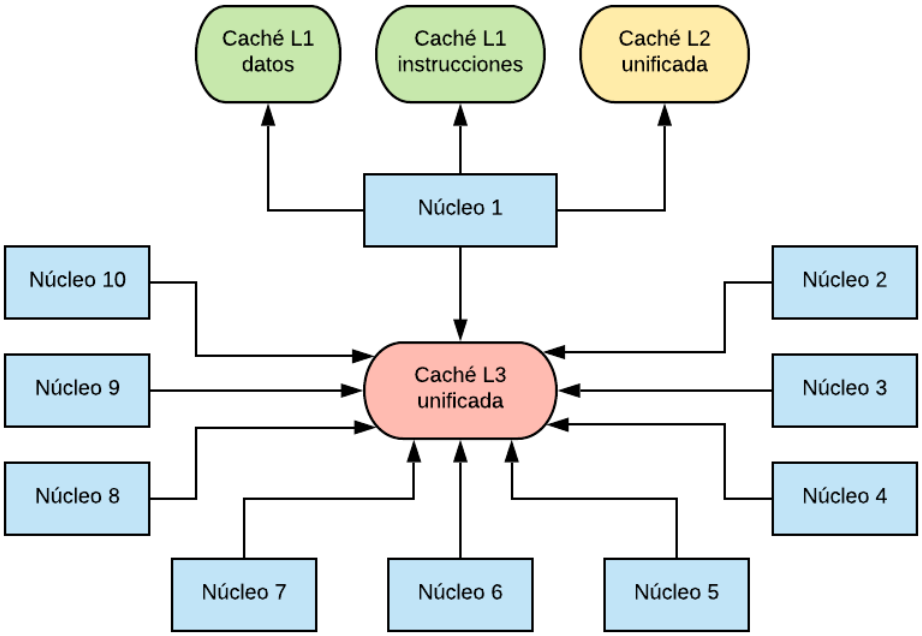

Figura 1: Esquema de la memoria caché

1.3. Experimentación

La experimentación fue gestionada con scripts del bash para facilitar el trabajo. El script desarrollado se ocupa de compilar todas las versiones del problema, ejecutarlas el número de veces necesario con distintos tamaños y para *OpenMP* con distintos hilos y de gestionar el uso de los registros con los resultados.

Código 2: script del bash de ejecución

```
1
2  #!/usr/bin/env bash
3
4  #Numero de cuaterniones del computo
5  q="100 10000 1000000 10000000"
6
7  #Numero de hilos para la version con OpenMP
8  hilos="1 2 4 8 16"
9
10 #Se halla el PID de la terminal actual y se le pasa a todos los codigos como
       semilla comun en esa ejecucion
11 pidbash=$$
12
13 #Se comprueba si se pasa algun argumento (no se acepta ninguno)
14 if test $# = 0
15 then
16   #Si el registro de la version 1 existe se borra (porque el codigo lo abre
         como append)
17   if test -f registro1.txt
18   then
19     rm -r registro1.txt
20   fi
21
22   #Si el registro de la version 1 con O2 existe se borra (porque el codigo lo
        abre como append)
23   if test -f registro1_opt.txt
24   then
25     rm -r registro1_opt.txt
26   fi
27
28   #Si el registro de la version 2 existe se borra (porque el codigo lo abre
         como append)
29   if test -f registro2.txt
30   then
31     rm -r registro2.txt
32   fi
33
34   #Si el registro de la version 3A existe se borra (porque el codigo lo abre
         como append)
35   if test -f registro3A.txt
36   then
37     rm -r registro3A.txt
38   fi
39
40   #Si el registro de la version 3B existe se borra (porque el codigo lo abre
         como append)
41   if test -f registro3B.txt
42   then
43     rm -r registro3B.txt
44   fi
45
46   #Si el registro de la version 4 existe se borra (porque el codigo lo abre
         como append)
47   if test -f registro4.txt
48   then
49     rm -r registro4.txt
```

```
50    fi
51
52    #Se compilan todas las versiones
53    icc -O0 ej1.c -Wall -o ej1
54    icc -O2 ej1.c -Wall -o ej1_optimizado
55    icc -O0 ej2.c -Wall -o ej2
56    icc -O0 ej3_a.c -Wall -o ej3_a
57    icc -O0 ej3_b.c -Wall -o ej3_b
58    icc -O0 ej4.c -Wall -o ej4 -qopenmp
59
60    #Se haran 10 pruebas con todos los programas
61    for i in `seq 1 10`
62    do
63      #En todos los registros se indica la prueba de la que se sacan los
          resultados
64      echo -e "PRUEBA $i\n" | tee -a registro1.txt registro1_opt.txt registro2.
          txt registro3A.txt registro3B.txt registro4.txt >> /dev/null
65
66      #Se ejecutan los codigos para todos los tama os de prueba
67      for j in $q
68      do
69        ./ej1 $j registro1.txt $pidbash
70        ./ej1_optimizado $j registro1_opt.txt $pidbash
71        ./ej2 $j $pidbash
72        ./ej3_a $j $pidbash
73        ./ej3_b $j $pidbash
74
75        #Se ejecuta la 4 version para cada tama o con todos los hilos
76        for k in $hilos
77        do
78          ./ej4 $j $k $pidbash
79        done
80      done
81
82      #Se a ade una separacion a los registros para que se lean mejor
83      echo "" | tee -a registro1.txt registro1_opt.txt registro2.txt registro3A
          .txt registro3B.txt registro4.txt >> /dev/null
84    done
85  else
86    echo "El numero de argumentos pasados al script no es valido"
87    exit 1
88  fi
```

Para poder cargar el script de ejecución en la cola del supercomputador fue necesario hacer otro script especial para ello que especificase como se iba a insertar. A continuación se muestra con comentarios el script de carga en la cola.

Código 3: script de carga para el CESGA

```
1
2  #!/bin/bash
3
4  #SBATCH -n 1                #Nodos en los que se ejecutara
5  #SBATCH -c 20               #Numero de hilos
6  #SBATCH -p shared           #La cola en la que se colocara
7  #SBATCH --qos=shared_short  #Tipo de proceso
```

```
#SBATCH -C has2s            #Procesador solicitado
#SBATCH -t 00:05:00         #Tiempo maximo de ejecucion

#Se carga el modulo de intel
module load intel

#Se le dan permisos al script de ejecucion
chmod 777 script.sh

#Se coloca el script de ejecucion en la cola
srun ./script.sh
```

2. Versión secuencial

Para la versión secuencial se emplearon los mecanismos básicos aportados por el lenguaje C, tales como bucles, punteros o arrays. Se realizó el cálculo haciendo uso de dos bucles, uno en el que se realiza la multiplicación y otro en el que se calcula el cuadrado de la multiplicación.

En esta versión, el cálculo del cuadrado de la multiplicación se hizo empleando la fórmula base de la multiplicación, sin realizar ninguna optimización, por lo que conllevará un mayor coste computacional en comparación con su versión optimizada.

2.1. Código

Código 4: código versión secuencial base

```

//---------------BIBLIOTECAS UTILIZADAS---------------

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

//---------------CONSTANTES---------------

#define LIMITE_SUP 100000
#define LIMITE_INF -100000

//---------------FUNCIONES EMPLEADAS---------------

//Programada por el equipo
void representar_quaternion(float* quaternion);

//Funciones externas de medida de ciclos
void access_counter(unsigned *hi, unsigned *lo);
void start_counter();
double get_counter();

//Variables globales para la medida de ciclos
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

//---------------FUNCION PRINCIPAL---------------

int main(int argc, char** argv)
```

```
30  {
31    FILE *fichero;
32    float *a, *b, *multiplicacion, reduccion[4];
33    int i, j, total, semilla = getpid();
34    double medidaCiclos;
35
36    //Se comprueba si el numero de argumentos es valido
37    //Se exige como minimo el numero de cuaterniones del computo y el nombre
          del fichero
38    if(argc >= 3 && argc <= 4)
39          {
40      //Se comprueba si el numero de cuaterniones es mayor que 0
41                  if((total = atoi(argv[1])) <= 0)
42            {
43              printf("El total de operaciones debe ser mayor que 0\n");
44              return 1;
45            }
46
47      //Se comprueba si la semilla de numeros aleatorios (opcional) es mayor
            que 0
48      if(argc == 4 && (semilla = atoi(argv[3])) < 0)
49      {
50              printf("La semilla debe ser positiva\n");
51              return 1;
52            }
53          }
54          else
55    {
56      printf("El numero de argumentos pasado no es valido\n");
57      return 1;
58    }
59
60    //Se establece la semilla, si no se indica por terminal se establece el PID
          del proceso como semilla
61    srand48(semilla);
62
63    //Se reserva memoria dinamicamente para los vectores de cuaterniones a,b y
          multiplicacion
64    a = (float*)malloc(total*4*sizeof(float));
65    b = (float*)malloc(total*4*sizeof(float));
66    multiplicacion = (float*)malloc(total*4*sizeof(float));
67
68    //Se inicializan los cuaterniones de a y b con numeros aleatorios en el
          rango definido por las constantes
69    for(i = 0; i < total; i++)
70      for(j = 0; j < 4; j++)
71      {
72        *(a + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
73        *(b + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
74      }
75
76    //Se inicia el cronometro de ciclos
77    start_counter();
78
79    //Cuaternion por cuaternion y componente a componente se aplica el producto
          Hamiltoniano
80    for(i = 0; i < total; i++)
```

```
81      {
82        *(multiplicacion + i*4 + 0) = *(a + i*4 + 0) * *(b + i*4 + 0) - *(a + i*4
              + 1) * *(b + i*4 + 1) - *(a + i*4 + 2) * *(b + i*4 + 2) - *(a + i*4 +
              3) * *(b + i*4 + 3);
83        *(multiplicacion + i*4 + 1) = *(a + i*4 + 0) * *(b + i*4 + 1) + *(a + i*4
              + 1) * *(b + i*4 + 0) + *(a + i*4 + 2) * *(b + i*4 + 3) - *(a + i*4 +
              3) * *(b + i*4 + 2);
84        *(multiplicacion + i*4 + 2) = *(a + i*4 + 0) * *(b + i*4 + 2) - *(a + i*4
              + 1) * *(b + i*4 + 3) + *(a + i*4 + 2) * *(b + i*4 + 0) + *(a + i*4 +
              3) * *(b + i*4 + 1);
85        *(multiplicacion + i*4 + 3) = *(a + i*4 + 0) * *(b + i*4 + 3) + *(a + i*4
              + 1) * *(b + i*4 + 2) - *(a + i*4 + 2) * *(b + i*4 + 1) + *(a + i*4 +
              3) * *(b + i*4 + 0);
86      }
87
88      //Se inicializa la variable de reduccion (el sumatorio)
89      for(i = 0; i < 4 ; i++)
90        reduccion[i] = 0.0;
91
92      //Con los valores de las multiplicaciones se calcula su cuadrado y se
            acumula en el sumatorio
93      for(i = 0; i < total; i++)
94      {
95        reduccion[0] += *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4 + 0)
              - *(multiplicacion + i*4 + 1) * *(multiplicacion + i*4 + 1) - *(
              multiplicacion + i*4 + 2) * *(multiplicacion + i*4 + 2) - *(
              multiplicacion + i*4 + 3) * *(multiplicacion+ i*4 + 3);
96        reduccion[1] += *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4 + 1)
              + *(multiplicacion + i*4 + 1) * *(multiplicacion + i*4 + 0) + *(
              multiplicacion + i*4 + 2) * *(multiplicacion + i*4 + 3) - *(
              multiplicacion + i*4 + 3) * *(multiplicacion + i*4 + 2);
97        reduccion[2] += *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4 + 2)
              - *(multiplicacion + i*4 + 1) * *(multiplicacion + i*4 + 3) + *(
              multiplicacion + i*4 + 2) * *(multiplicacion + i*4 + 0) + *(
              multiplicacion + i*4 + 3) * *(multiplicacion + i*4 + 1);
98        reduccion[3] += *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4 + 3)
              + *(multiplicacion + i*4 + 1) * *(multiplicacion + i*4 + 2) - *(
              multiplicacion + i*4 + 2) * *(multiplicacion + i*4 + 1) + *(
              multiplicacion + i*4 + 3) * *(multiplicacion + i*4 + 0);
99      }
100
101     //Se para el cronometro de ciclos
102     medidaCiclos = get_counter();
103
104     //Se muestra el resultado del sumatorio para ver si coincide con las otras
            versiones
105     printf("Resultado sumatorio: ");
106     representar_quaternion(reduccion);
107
108     //Se abre el fichero en modo append
109     if((fichero = fopen(argv[2], "a")) == NULL)
110     {
111       printf("Hubo un error al abrir el archivo en modo append\n");
112       return 1;
113     }
114
115     //Se escribe en el fichero el total de cuaterniones y la medida de ciclos
```

```
116     if(fprintf(fichero, "%d %f\n", total, medidaCiclos) < 0)
117     {
118       printf("Hubo un error al escribir el fichero\n");
119       return 1;
120     }
121
122     //Se cierra el fichero
123     if(fclose(fichero) == EOF)
124     {
125       printf("Hubo un error al cerrar el archivo\n");
126       return 1;
127     }
128
129     //Se libera la memoria de los vectores dinamicos utilizados
130     free(a);
131     free(b);
132     free(multiplicacion);
133
134     //Finalizacion normal del programa
135     return 0;
136   }
137
138   //Funcion para representar en pantalla un cuaternion
139   void representar_quaternion(float* quaternion)
140   {
141     printf("%f + %fi + %fj + %fk\n", *(quaternion + 0), *(quaternion + 1), *(
          quaternion + 2), *(quaternion + 3));
142   }
```

Código 5: compilación en terminal sin optimizaciones

```
icc -Wall -O0 version1.c -o version1

./version1 <tamVector> <semillaAleatorios>
```

Código 6: compilación en terminal con optimización O2

```
icc -Wall -O2 version1.c -o version1

./version1 <tamVector> <semillaAleatorios>
```

2.2. Resultados

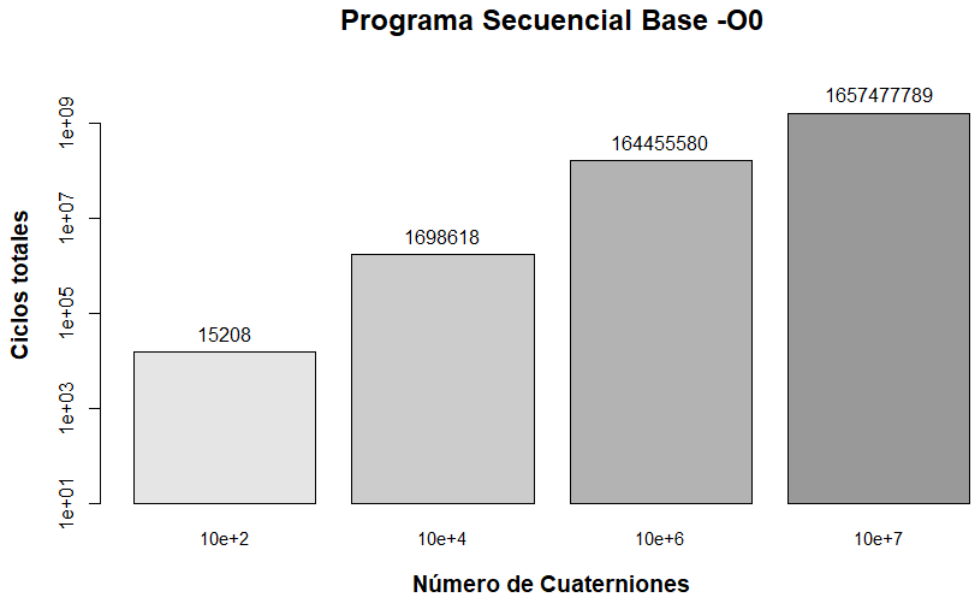

Figura 2: Resultados obtenidos con la opción de optimización -O0

Como era de esperarse. el número de ciclos obtenidos del código compilado con el nivel de optimización -O0 iba a ser bastante elevado en comparación a cuando se compile con el nivel -O2.

Cuando se emplea el nivel -O2, el compilador activará una serie de flags de optimización provocando una gran reducción de ciclos comparándolo con el caso anterior.

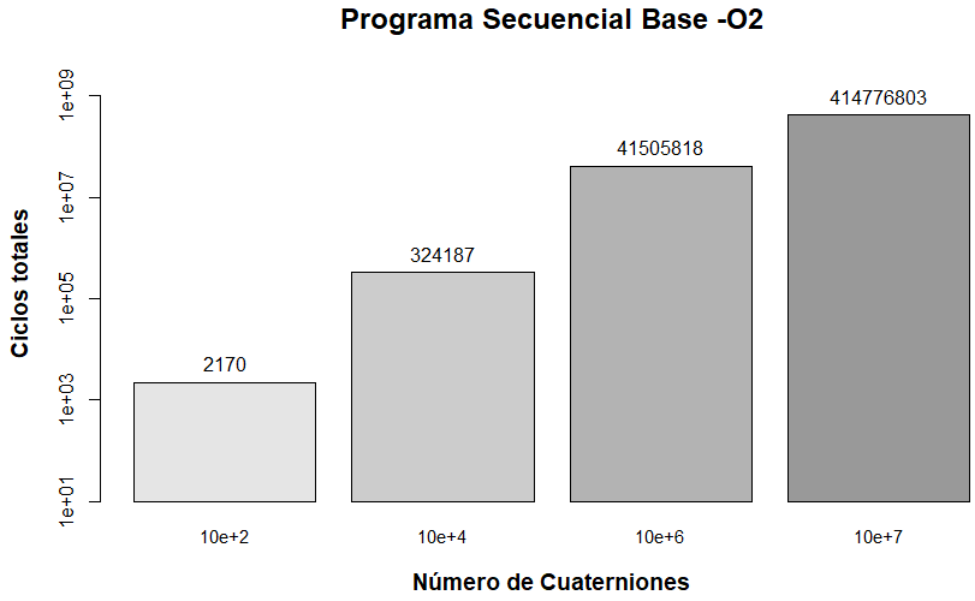

Figura 3: Resultados obtenidos con la opción de optimización -O2

3. Versión secuencial optimizada por el programador

3.1. Mejoras sobre la primera versión

Se plantean 3 grandes mejoras sobre el código original para conseguir que se ejecute de forma más eficiente. La primera medida consiste en simplificar el producto Hamiltoniano, cuando se tiene una potencia de cuaterniones se puede llegar a una versión del producto de un cuaternión por si mismo simplificado. Aquí está el producto original y que se sigue utilizando para la multiplicación auxiliar de a y b:

$$C_1 \cdot C_2 = a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i$$
$$+ (\mathrm{a}_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j + (a_1d_2 + b_1c_2 + c_1b_2 - d_1a_2)k$$

Sin embargo, cuando se hace el sumatorio de cuadrados se puede usar una expresión equivalente que requiere de muchas menos operaciones, y por lo tanto, de menos ciclos de cómputo:

$$C^2 = a^2 - b^2 - c^2 - d^2 + (2 \cdot ab)i + (2 \cdot ac)j + (2 \cdot ad)k$$

La segunda mejora consiste en fusionar los bucles de la multiplicación auxiliar y del sumatorio para ahorrar ciclos de cómputo. En la versión original primero se calcula el producto auxiliar de todos los quaterniones y después se eleva cada uno al cuadrado y se acumulan en un sumatorio, en la nueva versión a medida que se va calculando cada producto se le aplica el cuadrado y se empieza a acumular en el sumatorio, de forma que se ahorra un bucle.

Por último se consigue un ahorro de memoria notorio para cantidades grandes de cuaterniones al ahorrar la reserva dinámica del vector multiplicación. Como se han fusionado los bucles ahora solo es necesario guardar por cada iteración el resultado de una única multiplicación en vez de todas. Para el tamaño máximo de esta práctica esto supone un ahorro de 150 MB de memoria, una cifra considerable.

3.2. Código

Código 7: código versión secuencial optimizada por el programador

```
1
2  //---------------BIBLIOTECAS UTILIZADAS---------------
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7
8  //---------------CONSTANTES---------------
9
10 #define LIMITE_SUP 1000000
11 #define LIMITE_INF -1000000
12
13 //---------------FUNCIONES EMPLEADAS---------------
14
15 //Programada por el equipo
16 void representar_quaternion(float* quaternion);
17
```

```
18  //Funciones externas de medida de ciclos
19  void access_counter(unsigned *hi, unsigned *lo);
20  void start_counter();
21  double get_counter();
22
23  //Variables globales para la medida de ciclos
24  static unsigned cyc_hi = 0;
25  static unsigned cyc_lo = 0;
26
27  //---------------FUNCION PRINCIPAL---------------
28
29  int main(int argc, char** argv)
30  {
31    FILE *fichero;
32    float *a, *b, *multiplicacion, reduccion[] = {0.0, 0.0, 0.0, 0.0};
33    int i, j, total, semilla = getpid();
34    double medidaCiclos;
35
36    //Se comprueba si el numero de argumentos es valido se exige como minimo el
          numero de cuaterniones del computo
37    if(argc >= 2 && argc <= 3)
38          {
39      //Se comprueba si el numero de cuaterniones es mayor que 0
40                  if((total = atoi(argv[1])) <= 0)
41            {
42              printf("El total de operaciones debe ser mayor que 0\n");
43              return 1;
44            }
45
46      //Se comprueba si la semilla de numeros aleatorios (opcional) es mayor
            que 0
47      if(argc == 3 && (semilla = atoi(argv[2])) < 0)
48      {
49              printf("La semilla debe ser positiva\n");
50              return 1;
51            }
52          }
53          else
54    {
55      printf("El numero de argumentos pasado no es valido\n");
56      return 1;
57    }
58
59    //Se establece la semilla, si no se indica por terminal se establece el PID
          del proceso como semilla
60    srand48(semilla);
61
62    //Se reserva memoria dinamicamente para los vectores de cuaterniones a,b y
        multiplicacion
63    a = (float*)malloc(total*4*sizeof(float));
64    b = (float*)malloc(total*4*sizeof(float));
65    multiplicacion = (float*)malloc(total*4*sizeof(float));
66
67    //Se inicializan los cuaterniones de a y b con numeros aleatorios en el
        rango definido por las constantes
68    for(i = 0; i < total; i++)
69      for(j = 0; j < 4; j++)
```

```
70        {
71          *(a + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
72          *(b + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
73        }
74
75      //Se inicia el cronometro de ciclos
76      start_counter();
77
78      //En el mismo bucle se hace todo el computo
79      for(i = 0; i < total; i++)
80      {
81        //Primero se calcula la multiplicacion de a y b correspondiente
82        *(multiplicacion + i*4 + 0) = *(a + i*4 + 0) * *(b + i*4 + 0) - *(a + i*4
              + 1) * *(b + i*4 + 1) - *(a + i*4 + 2) * *(b + i*4 + 2) - *(a + i*4 +
              3) * *(b + i*4 + 3);
83        *(multiplicacion + i*4 + 1) = *(a + i*4 + 0) * *(b + i*4 + 1) + *(a + i*4
              + 1) * *(b + i*4 + 0) + *(a + i*4 + 2) * *(b + i*4 + 3) - *(a + i*4 +
              3) * *(b + i*4 + 2);
84        *(multiplicacion + i*4 + 2) = *(a + i*4 + 0) * *(b + i*4 + 2) - *(a + i*4
              + 1) * *(b + i*4 + 3) + *(a + i*4 + 2) * *(b + i*4 + 0) + *(a + i*4 +
              3) * *(b + i*4 + 1);
85        *(multiplicacion + i*4 + 3) = *(a + i*4 + 0) * *(b + i*4 + 3) + *(a + i*4
              + 1) * *(b + i*4 + 2) - *(a + i*4 + 2) * *(b + i*4 + 1) + *(a + i*4 +
              3) * *(b + i*4 + 0);
86
87        //Despues se acumula ya esa multiplicacion haciendo el producto y
              sumandola a la reduccion
88        reduccion[0] += *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4 + 0)
              - *(multiplicacion + i*4 + 1) * *(multiplicacion + i*4 + 1) - *(
              multiplicacion + i*4 + 2) * *(multiplicacion + i*4 + 2) - *(
              multiplicacion + i*4 + 3) * *(multiplicacion+ i*4 + 3);
89        reduccion[1] += 2 * *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4
              + 1);
90        reduccion[2] += 2 * *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4
              + 2);
91        reduccion[3] += 2 * *(multiplicacion + i*4 + 0) * *(multiplicacion + i*4
              + 3);
92      }
93
94      //Se para el cronometro de ciclos
95      medidaCiclos = get_counter();
96
97      //Se muestra el resultado del sumatorio para ver si coincide con las otras
            versiones
98      printf("Resultado sumatorio: ");
99      representar_quaternion(reduccion);
100
101     //Se abre el fichero en modo append
102     if((fichero = fopen("registro2.txt", "a")) == NULL)
103     {
104       printf("Hubo un error al abrir el archivo en modo append\n");
105       return 1;
106     }
107
108     //Se escribe en el fichero el total de cuaterniones y la medida de ciclos
109     if(fprintf(fichero, "%d %f\n", total, medidaCiclos) < 0)
110     {
```

```
111       printf("Hubo un error al escribir el fichero\n");
112       return 1;
113     }
114
115     //Se cierra el fichero
116     if(fclose(fichero) == EOF)
117     {
118       printf("Hubo un error al cerrar el archivo\n");
119       return 1;
120     }
121
122     //Se libera la memoria de los vectores dinamicos utilizados
123     free(a);
124     free(b);
125     free(multiplicacion);
126
127     //Finalizacion normal del programa
128     return 0;
129   }
130
131   //Funcion para representar en pantalla un cuaternion
132   void representar_quaternion(float* quaternion)
133   {
134     printf("%f + %fi + %fj + %fk\n", *(quaternion + 0), *(quaternion + 1), *(
          quaternion + 2), *(quaternion + 3));
135   }
```

Código 8: compilación en terminal sin optimizaciones

```
icc -Wall -O0 version2.c -o version2

./version1 <tamVector> <semillaAleatorios>
```

3.3. Resultados

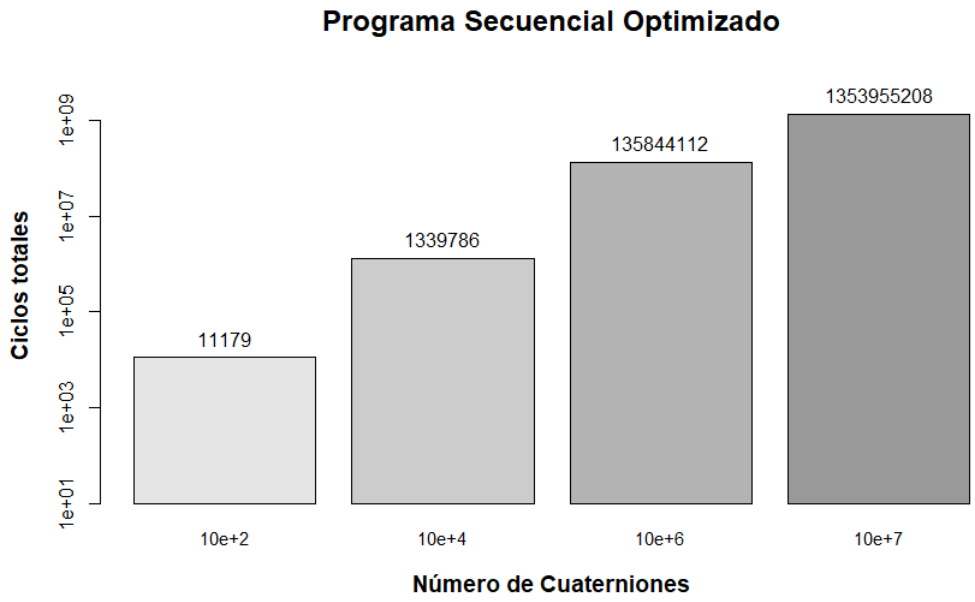

Figura 4: Resultados obtenidos con la versión optimizada

Al realizar la optimización señalada anteriormente, el número de ciclos se redujo con respecto a la versión anterior, cuando esta se compila con el nivel de optimización -O0, aunque no cuando se emplea el nivel -O2.

4. Versión con procesamiento SIMD usando la extensión SSE3

4.1. Vectorización de la multiplicación de cuaterniones

Para el cálculo de la multiplicación se emplearán las extensiones sse3, a mayores de las funciones explicadas en el documento del campus virtual, se utilizaron las funciones hsub, hadd y addsub.

Las funciones hsub y hadd, restan y suman respectivamente las compenentes 1 con la 2 y la componente 3 con la 4 de dos cuaterniones diferentes 2 a 2, de la siguiente forma:

$$hsub(a,b) = (a_1 - a_2) + (a_3 - a_4)i + (b_1 - b_2)j + (b_3 - b_4)k$$

$$hadd(a,b) = (a_2 + a_1) + (a_4 + a_3)i + (b_2 + b_1)j + (b_4 + b_3)k$$

En el caso del hadd, el orden es al revés, hecho que no influye ya que la suma tiene la propiedad conmutativa, por lo que no influye el orden de los operandos.

La función addsub, se encarga de realizar sumas y restas según la posición del operando, resta en las posiciones pares y suma en las impares, esto ocurre de la siguiente manera:

$$addsub(a,b) = (a_1 - b_1) + (a_2 + b_2)i + (a_3 - b_3)j + (a_4 + b_4)k$$

Las posiciones pares e impares se corresponden con la notación informática pues el conteo de la posición se iniciaría en 0, por lo que las posiciones pares serán la primera y la tercera

El procedimiento empleado, para el cálculo de la multiplicación, fue el siguiente:

- Primero se obtuvieron los operandos deseados, para ello se realizan una serie de shuffles sobre el operando b de la multiplicación, obteniéndose los siguientes cuaterniones:

$$b1221 = b_1 + b_2 i + b_2 j + b_1 k$$

$$b2332 = b_2 + b_3 i + b_3 j + b_2 k$$

$$b2112 = b_2 + b_1 i + b_1 j + b_2 k$$

$$b3223 = b_3 + b_2 i + b_2 j + b_3 k$$

- Posteriormente, mediante el uso de mul y de hadd/hsub, se obtienen las mitades de cada una de las componentes de la multiplicación por separado, siendo necesario unirlas en el paso posterior, en este paso se obtienen los siguientes resultados:

$$ParteResta = (a_1 * b_1 - a_2 * b_2) + (a_3 * b_2 - a_4 * b_1)i + (a_1 * b_3 - a_2 * b_4)j + (a_3 * b_4 - a_4 * b_3)k$$

$$ParteSuma = (a_1 * b_2 + a_2 * b_1) + (a_3 * b_1 + a_4 * b_2)i + (a_1 * b_4 + a_2 * b_3)j + (a_3 * b_3 + a_4 * b_4)k$$

- El resultado de la multiplicacion se obtendrá aplicando un par de shuffles sobre parteResta y parteSuma, para ordenar los componentes de la forma apropiada y usando un addsub se unen ambas partes, obteniendo el siguiente cuaternión:

$$multiplicacion = (a_1 * b_1 - a_2 * b_2 - (a_3 * b_3 + a_4 * b_4)) + (a_1 * b_3 - a_2 * b_4 + a_3 * b_1 + a_4 * b_2)i +$$

$$(a_1 * b_4 + a_2 * b_3 - (a_3 * b_2 - a_4 * b_1))j + (a_1 * b_2 + a_2 * b_1 + a_3 * b_4 - a_4 * b_3)k$$

- Finalmente, el resultado correcto se obtendrá aplicando un shuffle a la multiplicación anterior, para conseguir el orden final, obteniendo:

$$multiplicacion = (a_1 * b_1 - a_2 * b_2 - (a_3 * b_3 + a_4 * b_4)) + (a_1 * b_2 + a_2 * b_1 + a_3 * b_4 - a_4 * b_3)i +$$

$$(a_1 * b_3 - a_2 * b_4 + a_3 * b_1 + a_4 * b_2)j + (a_1 * b_4 + a_2 * b_3 - (a_3 * b_2 - a_4 * b_1))k$$

- Para obtener la acumulación del cuadrado de la multiplicación, primero se multiplicará el registro multiplicación consigo mismo, también se emplea un cuaternión con valor -1 en todas las componentes menos en la primera, y se acumularán todas las componentes en la primera realizando dos llamadas consecutivas a hadd, dando como resultado:

$$resultado_1 = mul_1 * mul_1 - mul_2 * mul_2 - mul_3 * mul_3 - mul_4 * mul_4$$

- En el segundo paso, se usará un shuffle para obtener un cuaternión cuyas componentes sean todas iguales a la primera componente de la multiplicacion, de forma que después se pueda multiplicar por la el cuaternión multiplicación y se obtenga el resto de componentes de resultado, dando lugar a :

$$resultado_2 = mul_1 * mul_2$$

$$resultado_3 = mul_1 * mul_3$$

$$resultado_4 = mul_1 * mul_4$$

- Como último paso, se realizan una serie de shuffle para reordenar el registro y se acumula su valor usando un add. Para finalizar, se multiplicarán las tres últimas componentes por 2, una vez se salga del bucle.Por lo tanto, el resultado final será:

$$resultado = (mul_1 * mul_1 - mul_2 * mul_2 - mul_3 * mul_3 - mul_4 * mul_4) +$$

$$(mul_1 * mul_2 * 2)i + (mul_1 * mul_3 * 2)j + (mul_1 * mul_4 * 2)k$$

4.1.1. Código

Código 9: código versión con vectorización de la multiplicación

```
1
2  //---------------BIBLIOTECAS UTILIZADAS---------------
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <pmmintrin.h>
8
9  //---------------CONSTANTES---------------
10
11 #define LIMITE_SUP 1000000
12 #define LIMITE_INF -1000000
13
14 //---------------FUNCIONES EMPLEADAS---------------
15
```

```
16  //Programada por el equipo
17  void representar_quaternion(float* quaternion);
18
19  //Funciones externas de medida de ciclos
20  void access_counter(unsigned *hi, unsigned *lo);
21  void start_counter();
22  double get_counter();
23
24  //Variables globales para la medida de ciclos
25  static unsigned cyc_hi = 0;
26  static unsigned cyc_lo = 0;
27
28  //---------------FUNCION PRINCIPAL---------------
29
30  int main(int argc, char** argv)
31  {
32          FILE *fichero;
33          int i, total, semilla = getpid();
34          float *x, *y;
35          float k[4] __attribute__((aligned(16)));
36          double medidaCiclos;
37
38          //Se comprueba si el numero de argumentos es valido se exige como
                minimo el numero de cuaterniones del computo
39          if(argc >= 2 && argc <= 3)
40          {
41                  //Se comprueba si el numero de cuaterniones es mayor que 0
42          if((total = atoi(argv[1])) <= 0)
43          {
44            printf("El total de operaciones debe ser mayor que 0\n");
45            return 1;
46          }
47
48                  //Se comprueba si la semilla de numeros aleatorios (opcional)
                        es mayor que 0
49          if(argc == 3 && (semilla = atoi(argv[2])) < 0)
50          {
51              printf("La semilla debe ser positiva\n");
52              return 1;
53            }
54          }
55          else
56          {
57              printf("El numero de argumentos pasado no es valido\n");
58              return 1;
59          }
60
61          //Se establece la semilla, si no se indica por terminal se establece
                el PID del proceso como semilla
62          srand48(semilla);
63
64          __m128 b1221, b3443, b2112, b4334, parteResta, parteSuma, Cuadrado,
                _0000, Cons0, Cons1, Cons2, ResParc, Producto, Resultado;
65          __m128 *a, *b, *suma, *multiplicacion;
66
67          //Se reserva memoria dinamicamente para los vectores de cuaterniones
                a, b, suma y multiplicacion
```

```
68          a = (__m128*)_mm_malloc(total*sizeof(__m128), 16);
69          b = (__m128*)_mm_malloc(total*sizeof(__m128), 16);
70          suma = (__m128*)_mm_malloc(total*sizeof(__m128), 16);
71          multiplicacion = (__m128*)_mm_malloc(total*sizeof(__m128), 16);
72
73          //Se reserva memoria dinamicamente para x e y
74          //Ambas variables deben estar alineadas a 16 bytes
75          x = (float*)_mm_malloc(4 * sizeof(float),16);
76
77          y = (float*)_mm_malloc(4 * sizeof(float),16);
78
79          for(i = 0; i < total; i++)
80      //Se realiza un bucle para obtener todos los valores aleatorios necesarios
            para realizar los calculos
81          {
82             *(x + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
83             *(y + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
84             *(x + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
85             *(y + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
86             *(x + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
87             *(y + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
88             *(x + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
89             *(y + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
90
91          //Finalmente se carga el valor en una posicion del cuaternion
92             *(a + i) = _mm_load_ps(&x[0]);
93             *(b + i) = _mm_load_ps(&y[0]);
94          }
95
96          Cons0 = _mm_set1_ps(0.0);
97          Cons1 = _mm_set_ps(-1.0,-1.0,-1.0,1.0);
98          //Cargamos tres registros de 128 bits con valores constantes, los
               cuales utilizaremos para calculos posteriores
99          Cons2 = _mm_set_ps(2.0,2.0,2.0,1.0);
100
101         //Se inicia el cronometro de ciclos
102         start_counter();
103
104         Resultado = _mm_setzero_ps(); //Inicializamos el registro resultado
105
106         for(i = 0; i < total; i++)
107         {
108
109                 b1221 = _mm_shuffle_ps(b[i],b[i],_MM_SHUFFLE(0,1,1,0));
110                 b3443 = _mm_shuffle_ps(b[i],b[i],_MM_SHUFFLE(2,3,3,2));
111
112                 //Realizamos una serie de shuffles para obtener los operandos
                        deseados de la componente de la multiplicacion b
113                 b2112 = _mm_shuffle_ps(b[i],b[i],_MM_SHUFFLE(1,0,0,1));
114                 b4334 = _mm_shuffle_ps(b[i],b[i],_MM_SHUFFLE(3,2,2,3));
115
116                 parteResta = _mm_hsub_ps(_mm_mul_ps(a[i],b1221),_mm_mul_ps(a[
                        i],b3443));
117
118                 //Empleando las funciones hsub y hadd vamos restando/sumando
                        las componentes de la multiplicacion entre si 2 a 2
```

```
119                     parteSuma = _mm_hadd_ps(_mm_mul_ps(a[i],b2112),_mm_mul_ps(a[i
                            ],b4334));
120
121                     //Realizamos un addsub de forma que se alterna entre una
                            resta(en las posiciones pares) y una suma(en las
                            posiciones impares), combinados con suffles para obtener
                            el orden deseado
122                     multiplicacion[i] = _mm_addsub_ps(_mm_shuffle_ps(parteResta,
                            parteSuma,_MM_SHUFFLE(0,2,2,0)),_mm_shuffle_ps(parteSuma,
                            parteResta,_MM_SHUFFLE(3,1,1,3)));
123                     multiplicacion[i] = _mm_shuffle_ps(multiplicacion[i],
                            multiplicacion[i],_MM_SHUFFLE(2,1,3,0));
124
125                     //Finalmente realizamos un shuffle final para obtener el
                            resultado final de la multiplicaion
126
127                     _0000 = _mm_shuffle_ps(multiplicacion[i],multiplicacion[i],
                            _MM_SHUFFLE(0,0,0,0));
128
129                     //Obtenemos un registro que solo contiene la primera posicion
                             de la multiplicacion
130
131                     Cuadrado = _mm_hadd_ps(_mm_hadd_ps(_mm_mul_ps(_mm_mul_ps(
                            multiplicacion[i],multiplicacion[i]),Cons1),Cons0),Cons0);
                             //Usando dos veces hadd combinamos en una unica
                            componente el cuadrado de la resta de los elementos de la
                            multiplicacion
132                     Producto = _mm_mul_ps(_0000,multiplicacion[i]);
133
134                     //Calculamos las otras tres componentes del cuadrado de la
                            multiplicacion
135                     ResParc = _mm_shuffle_ps(_mm_shuffle_ps(Cuadrado,Producto,
                            _MM_SHUFFLE(1,1,0,0)),Producto,_MM_SHUFFLE(3,2,2,0));
136                     // Para finalizar obtenemos el resultado correcto reordenando
                             el registro usando shuffles
137
138                     Resultado = _mm_add_ps(Resultado,ResParc);
139                     //Sumamos el resultado parcial obtenido antes al resultado
                            total
140     }
141
142         Resultado = _mm_mul_ps(Resultado,Cons2);
143         //Multiplicamos las ultimas tres componentes del cuaternion por dos
144
145         //Se para el cronometro de ciclos
146         medidaCiclos = get_counter();
147
148         _mm_store_ps(&k[0], Resultado);
149
150         //Se muestra el resultado del sumatorio para ver si coincide con las
                otras versiones
151         printf("Resultado sumatorio: ");
152         representar_quaternion(k);
153
154         //Se abre el fichero en modo append
155         if((fichero = fopen("registro3A.txt", "a")) == NULL)
156         {
```

```
157                 printf("Hubo un error al abrir el archivo en modo append\n");
158                 return 1;
159             }
160 
161             //Se escribe en el fichero el total de cuaterniones y la medida de
                    ciclos
162             if(fprintf(fichero, "%d %f\n", total, medidaCiclos) < 0)
163             {
164                 printf("Hubo un error al escribir el fichero\n");
165                 return 1;
166             }
167 
168             //Se cierra el fichero
169             if(fclose(fichero) == EOF)
170             {
171                 printf("Hubo un error al cerrar el archivo\n");
172                 return 1;
173             }
174 
175             //Se libera la memoria de los vectores dinamicos utilizados
176             _mm_free(a);
177             _mm_free(b);
178             _mm_free(suma);
179             _mm_free(multiplicacion);
180             _mm_free(x);
181             _mm_free(y);
182 
183             //Finalizacion normal del programa
184             return 0;
185     }
186 
187     //Funcion para representar en pantalla un cuaternion
188     void representar_quaternion(float* quaternion)
189     {
190         printf("%f, %fi, %fj, %fk\n", quaternion[0], quaternion[1], quaternion
                [2], quaternion[3]);
191     }
```

Código 10: compilación en terminal sin optimizaciones

```
icc -Wall -O0 version3A.c -o version3A

./version1 <tamVector> <semillaAleatorios>
```

4.1.2. Resultados

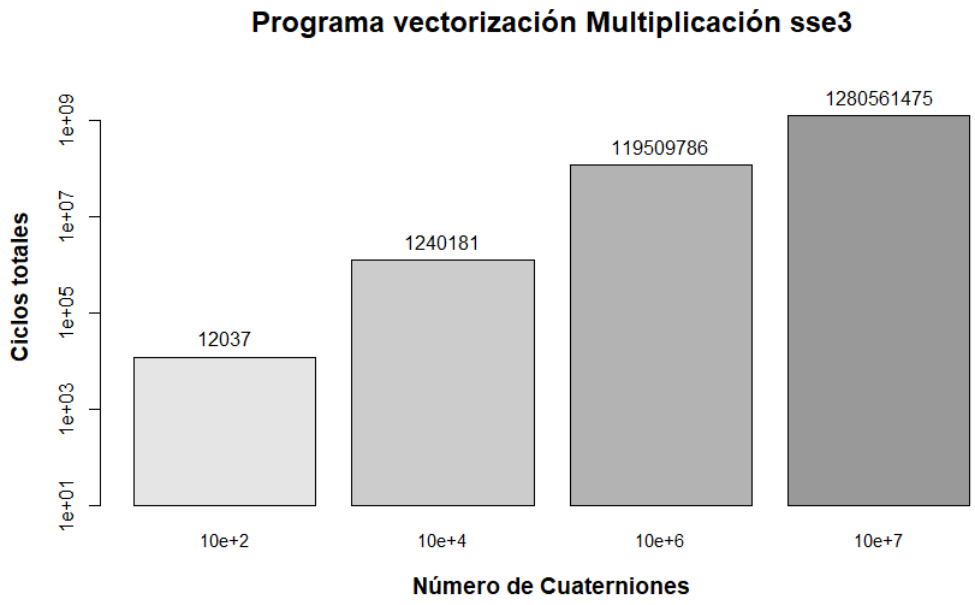

Figura 5: Resultados obtenidos al vectorizar la multiplicación usando sse3

Al emplear la extensión sse3 se pudo reducir en parte el número de ciclos obtenidos con respecto a las dos versiones anteriores, cuando ambas se compilan con el nivel de optimización -O0.

4.2. Vectorización de las iteraciones del bucle

Para realizar la vectorización de las iteraciones del bucle, en el que se calcula la multiplicación de cuaterniones, se emplearon 8 registros de 128 bits , de forma que, a cada uno de ellos le corresponda una de las componentes de los operandos, habiendo 4 registros para el cuaternión a y 4 registros para el cuaternión b.

De esta forma, se dispone, de los registros A1(componentes reales del cuaternión a), A2(componentes i del cuaternión a), A3(componentes j del cuaternión a) y A4(componentes k del cuaternión k), y de igual forma, para las componenentes de b.

Además, también se tratan, de la misma forma los registros de multiplicación y de resultado, teniendo un registro para cada componente.

Para el cálculo de la multiplicación, entonces se utilizaran los componentes de los cuaterniones de forma independiente, tomando como ejemplo la primera componente de la multiplicación :

$$multiplicación_0 = a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4$$

El proceso que realizaríamos sería:

$$multiplicación[0] = A1[i] * B1[i] - A2[i] * B2[i] - A3[i] * B3[i] - A4[i] * B4[i]$$

Así, logramos que en cada iteración del bucle se calcule la multiplicación de cuatro cuaterniones, en lugar de solo uno, maximizando el número de operaciones que podemos realizar y reduciendo consderablemente el número de ciclos empleado.

El cálculo de la acumulación del cuadrado de la multiplicación sigue un proceso igual, ya que como explicamos con anterioridad, los registros multiplicación y resultado también cuentan con un registro por cada componente, utilizando las componentes de forma independiente, optimizando también el cálculo de la acumulación. Aunque para finalizar su cálculo se emplearán un hadd por componente y tres shuffles necesarios para juntar y reordenar las componentes de resultado en un único registro llamado ResFinal.

4.2.1. Código

Código 11: código versión con vectorización de las iteraciones del bucle

```
1
2  //---------------BIBLIOTECAS UTILIZADAS---------------
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <pmmintrin.h>
8
9  //---------------CONSTANTES---------------
10
11 #define LIMITE_SUP 1000000
12 #define LIMITE_INF -1000000
13
14 //---------------FUNCIONES EMPLEADAS---------------
15
16 //Programada por el equipo
17 void representar_quaternion(float* quaternion);
18
19 //Funciones externas de medida de ciclos
20 void access_counter(unsigned *hi, unsigned *lo);
21 void start_counter();
22 double get_counter();
23
24 //Variables globales para la medida de ciclos
25 static unsigned cyc_hi = 0;
26 static unsigned cyc_lo = 0;
27
28 //---------------FUNCION PRINCIPAL---------------
29
30 int main(int argc, char** argv)
31 {
32         FILE *fichero;
33         int i, total, semilla = getpid();
34         double medidaCiclos;
35         float a1[4] __attribute__((aligned(16))),
36               a2[4] __attribute__((aligned(16))),
37               a3[4] __attribute__((aligned(16))),
38               a4[4] __attribute__((aligned(16))),
39               b1[4] __attribute__((aligned(16))),
40               b2[4] __attribute__((aligned(16))),
41               b3[4] __attribute__((aligned(16))),
42               b4[4] __attribute__((aligned(16))),
43               x[4] __attribute__((aligned(16)));
44
```

```
45          //Se comprueba si el numero de argumentos es valido se exige como
                minimo el numero de cuaterniones del computo
46          if(argc >= 2 && argc <= 3)
47          {
48                  //Se comprueba si el numero de cuaterniones es mayor que 0
49              if((total = atoi(argv[1])) <= 0)
50              {
51                  printf("El total de operaciones debe ser mayor que 0\n");
52                  return 1;
53              }
54
55                  //Se comprueba si la semilla de numeros aleatorios (opcional)
                        es mayor que 0
56              if(argc == 3 && (semilla = atoi(argv[2])) < 0)
57              {
58                      printf("La semilla debe ser positiva\n");
59                      return 1;
60              }
61          }
62          else
63          {
64              printf("El numero de argumentos pasado no es valido\n");
65              return 1;
66          }
67
68          //Se establece la semilla, si no se indica por terminal se establece
                el PID del proceso como semilla
69          srand48(semilla);
70
71          __m128 Cons0, Cons2, ResFinal;
72          __m128 *A1,*A2,*A3,*A4,*B1,*B2,*B3,*B4, *multiplicacion, *Resultado;
73
74          //Se reserva memoria dinamicamente para los vectores de cuaterniones
                A1,A2,A3,A4,B1,B2,B3,B4, suma y multiplicacion
75          A1 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
76          B1 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
77          A2 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
78          B2 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
79          A3 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
80          B3 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
81          A4 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
82          B4 = (__m128*)_mm_malloc((total/4)*sizeof(__m128), 16);
83          multiplicacion = (__m128*)_mm_malloc(4*sizeof(__m128), 16);
84          Resultado = (__m128*)_mm_malloc(4*sizeof(__m128), 16);
85
86      //Se llenan los float correspondientes con cada uno de los valores
            aleatorios necesario, se emplea este orden para que los resultados con
             el resto de versiones sea el mismo
87          for(i = 0; i < total/4; i++)
88          {
89          *(a1 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
90          *(b1 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
91          *(a2 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
92          *(b2 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
93          *(a3 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
94          *(b3 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
95          *(a4 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
```

```
96          *(b4 + 0) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
97
98          *(a1 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
99          *(b1 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
100         *(a2 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
101         *(b2 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
102         *(a3 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
103         *(b3 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
104         *(a4 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
105         *(b4 + 1) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
106
107         *(a1 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
108         *(b1 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
109         *(a2 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
110         *(b2 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
111         *(a3 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
112         *(b3 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
113         *(a4 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
114         *(b4 + 2) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
115
116         *(a1 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
117         *(b1 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
118         *(a2 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
119         *(b2 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
120         *(a3 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
121         *(b3 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
122         *(a4 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
123         *(b4 + 3) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
124
125     //Se carga cada uno de los registros correspondientes a cada una de las
        componentes de los cuaterniones
126         *(A1 + i) = _mm_load_ps(&a1[0]);
127         *(B1 + i) = _mm_load_ps(&b1[0]);
128         *(A2 + i) = _mm_load_ps(&a2[0]);
129         *(B2 + i) = _mm_load_ps(&b2[0]);
130         *(A3 + i) = _mm_load_ps(&a3[0]);
131         *(B3 + i) = _mm_load_ps(&b3[0]);
132         *(A4 + i) = _mm_load_ps(&a4[0]);
133         *(B4 + i) = _mm_load_ps(&b4[0]);
134         }
135
136         Cons0 = _mm_set1_ps(0.0);
137         Cons2 = _mm_set1_ps(2.0);
138         //Se inicializan una serie de constantes necesarias para el calculo
139
140         //Se inicia el cronometro de ciclos
141         start_counter();
142
143         //Se inicializa a 0 el registro Resultado
144         Resultado[0] = _mm_setzero_ps();
145         Resultado[1] = _mm_setzero_ps();
146         Resultado[2] = _mm_setzero_ps();
147         Resultado[3] = _mm_setzero_ps();
148
149         for(i = 0; i < total/4; i++)
150         {
151
```

```
152             //Combinando sub, add y mul realizamos las operaciones componente por
                    componente de los cuaterniones, de forma que  obtenemos las
                componentes resultantesde la multiplicacion
153                 multiplicacion[0] = _mm_sub_ps(_mm_sub_ps(_mm_sub_ps(
                        _mm_mul_ps(A1[i],B1[i]),_mm_mul_ps(A2[i],B2[i])),
                        _mm_mul_ps(A3[i],B3[i])),_mm_mul_ps(A4[i],B4[i]));
154
155                 multiplicacion[1] = _mm_sub_ps(_mm_add_ps(_mm_add_ps(
                        _mm_mul_ps(A1[i],B2[i]),_mm_mul_ps(A2[i],B1[i])),
                        _mm_mul_ps(A3[i],B4[i])),_mm_mul_ps(A4[i],B3[i]));
156
157                 multiplicacion[2] = _mm_add_ps(_mm_add_ps(_mm_sub_ps(
                        _mm_mul_ps(A1[i],B3[i]),_mm_mul_ps(A2[i],B4[i])),
                        _mm_mul_ps(A3[i],B1[i])),_mm_mul_ps(A4[i],B2[i]));
158
159                 multiplicacion[3] = _mm_add_ps(_mm_sub_ps(_mm_add_ps(
                        _mm_mul_ps(A1[i],B4[i]),_mm_mul_ps(A2[i],B3[i])),
                        _mm_mul_ps(A3[i],B2[i])),_mm_mul_ps(A4[i],B1[i]));
160
161                 //Posteriormente realizamos el mismo proceso(combinando sub,
                        add,mul) para calcular el cuadrado de la multiplicacion y
                        sumarlo a cada componenente del resultado
162                 Resultado[0] = _mm_add_ps(Resultado[0],_mm_sub_ps(_mm_sub_ps(
                        _mm_sub_ps(_mm_mul_ps(multiplicacion[0],multiplicacion[0])
                        ,_mm_mul_ps(multiplicacion[1],multiplicacion[1])),
163                                     _mm_mul_ps(multiplicacion[2],multiplicacion
                                            [2])),_mm_mul_ps(multiplicacion[3],
                                            multiplicacion[3])));
164
165                 Resultado[1] = _mm_add_ps(Resultado[1],_mm_mul_ps(
                        multiplicacion[0],multiplicacion[1]));
166
167                 Resultado[2] = _mm_add_ps(Resultado[2],_mm_mul_ps(
                        multiplicacion[0],multiplicacion[2]));
168
169                 Resultado[3] = _mm_add_ps(Resultado[3],_mm_mul_ps(
                        multiplicacion[0],multiplicacion[3]));
170     }
171
172     //Multiplicamos las tres ultimas componentes del resultado por 2
173         Resultado[1] = _mm_mul_ps(Resultado[1],Cons2);
174
175         Resultado[2] = _mm_mul_ps(Resultado[2],Cons2);
176
177         Resultado[3] = _mm_mul_ps(Resultado[3],Cons2);
178
179
180         //Para finalizar, acumulamos todas las componentes del resultado en
                una y realizando tres shuffles reordenamos todo para obtener el
                resultado final
181         Resultado[0] = _mm_hadd_ps(_mm_hadd_ps(Resultado[0],Cons0),Cons0);
182
183         Resultado[1] = _mm_hadd_ps(_mm_hadd_ps(Resultado[1],Cons0),Cons0);
184
185         Resultado[2] = _mm_hadd_ps(_mm_hadd_ps(Resultado[2],Cons0),Cons0);
186
187         Resultado[3] = _mm_hadd_ps(_mm_hadd_ps(Resultado[3],Cons0),Cons0);
```

```
188
189         ResFinal = _mm_shuffle_ps(_mm_shuffle_ps(Resultado[1],Resultado[0],
                _MM_SHUFFLE(0,0,0,0)),_mm_shuffle_ps(Resultado[3],Resultado[2],
                _MM_SHUFFLE(0,0,0,0)),_MM_SHUFFLE(0,2,0,2));
190
191         //Se para el cronometro de ciclos
192         medidaCiclos = get_counter();
193
194     //Se muestra el resultado del sumatorio para ver si coincide con las
            otras versiones
195         _mm_store_ps(&x[0], ResFinal);
196         printf("Resultado: ");
197         representar_quaternion(x);
198
199
200         //Se abre el fichero en modo append
201         if((fichero = fopen("registro3B.txt", "a")) == NULL)
202         {
203             printf("Hubo un error al abrir el archivo en modo append\n");
204             return 1;
205         }
206
207         //Se escribe en el fichero el total de cuaterniones y la medida de
                ciclos
208         if(fprintf(fichero, "%d %f\n", total, medidaCiclos) < 0)
209         {
210             printf("Hubo un error al escribir el fichero\n");
211             return 1;
212         }
213
214         //Se cierra el fichero
215         if(fclose(fichero) == EOF)
216         {
217             printf("Hubo un error al cerrar el archivo\n");
218             return 1;
219         }
220
221         //Se libera la memoria de los vectores dinamicos utilizados
222         _mm_free(A1);
223         _mm_free(B1);
224         _mm_free(A2);
225         _mm_free(B2);
226         _mm_free(A3);
227         _mm_free(B3);
228         _mm_free(A4);
229         _mm_free(B4);
230         _mm_free(multiplicacion);
231         _mm_free(Resultado);
232
233         //Finalizacion normal del programa
234         return 0;
235 }
236
237 //Funcion para representar en pantalla un cuaternion
238 void representar_quaternion(float* quaternion)
239 {
```

```
240         printf("%f, %fi, %fj, %fk\n", quaternion[0], quaternion[1], quaternion
                [2], quaternion[3]);
241     }
```

Código 12: compilación en terminal sin optimizaciones

```
icc -Wall -O0 version3B.c -o version3B

./version1 <tamVector> <semillaAleatorios>
```

4.2.2. Resultados

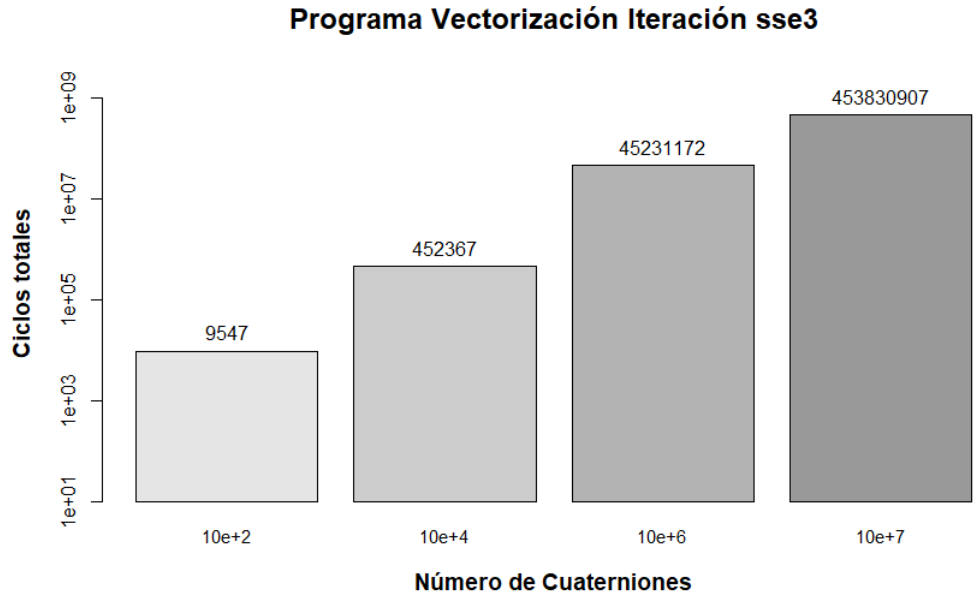

Figura 6: Resultados ob tenidos al vectorizar cada iteración de la multiplicación usando sse3

En comparación con la anterior versión, se puede ver, una gran mejoría puesto que en esta versión se optimizan los cálculos de forma que no se malgastan ciclos realizando calculos innecesarios.

Por lo que, los ciclos obtenidos para cualquier tamaño son menores en comparación con la versión anterior

5. Versión con paralelización de hilos usando OpenMP

La última versión del código consiste en resolver el cálculo usando hilos con la API *OpenMP*. La creación y gestión de hilos es muy costosa para el computador y para cantidades pequeñas de cálculo va a ser contraproducente hacer uso de esta técnica ya que el coste va a superar a las ganancias de tiempo. Se observará que para grandes cómputos va a dar los mejores resultados con muchos hilos y para cantidades pequeñas va a ser la peor opción.

La gestión de hilos supone un gran problema a la hora de optimizar el rendimiento del código. El equipo tuvo que enfrentarse a problemas como minimizar el número de accesos a variables compartidas,

asegurar el acceso exclusivo a las variables compartidas y evitar el uso de *pragmas* como *critical* que elevarían el coste enormemente

5.1. Código

Código 13: código versión multihilo con OpenMP

```
1
2  //---------------BIBLIOTECAS UTILIZADAS---------------
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <omp.h>
8
9  //---------------CONSTANTES---------------
10
11 #define LIMITE_SUP 1000000
12 #define LIMITE_INF -1000000
13
14 //---------------FUNCIONES EMPLEADAS---------------
15
16 //Programada por el equipo
17 void representar_quaternion(float* quaternion);
18
19 //Funciones externas de medida de ciclos
20 void access_counter(unsigned *hi, unsigned *lo);
21 void start_counter();
22 double get_counter();
23
24 //Variables globales para la medida de ciclos
25 static unsigned cyc_hi = 0;
26 static unsigned cyc_lo = 0;
27
28 //---------------FUNCION PRINCIPAL---------------
29
30 int main(int argc, char** argv)
31 {
32   FILE *fichero;
33   float *a, *b, reduccion[] = {0.0, 0.0, 0.0, 0.0};
34   int i, j, total, numeroHilos, semilla = getpid();
35   double medidaCiclos;
36
37   //Se comprueba si el numero de argumentos es valido se exige como minimo el
         numero de cuaterniones del computo y el numero de hilos
38   if(argc >= 3 && argc <= 4)
39   {
40     //Se comprueba si el numero de cuaterniones es mayor que 0
41     if((total = atoi(argv[1])) <= 0)
42     {
43       printf("El total de operaciones debe ser mayor que 0\n");
44       return 1;
45     }
46
47     //Se comprueba si el numero de hilos es mayor que 0
48     if((numeroHilos = atoi(argv[2])) <= 0)
```

```
49        {
50          printf("El numero de hilos debe ser mayor que 0\n");
51          return 1;
52        }
53
54        //Se comprueba si la semilla de numeros aleatorios (opcional) es mayor
              que 0
55        if(argc == 4 && (semilla = atoi(argv[3])) < 0)
56        {
57           printf("La semilla debe ser positiva\n");
58           return 1;
59        }
60      }
61      else
62      {
63        printf("El numero de argumentos pasado no es valido\n");
64        return 1;
65      }
66
67      //Se establece la semilla, si no se indica por terminal se establece el PID
            del proceso como semilla
68      srand48(semilla);
69
70      //Se reserva memoria dinamicamente para los vectores de cuaterniones a,b y
            multiplicacion
71      a = (float*)malloc(total*4*sizeof(float));
72      b = (float*)malloc(total*4*sizeof(float));
73
74      //Se inicializan los cuaterniones de a y b con numeros aleatorios en el
            rango definido por las constantes
75      for(i = 0; i < total; i++)
76        for(j = 0; j < 4; j++)
77        {
78          *(a + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
79          *(b + i*4 + j) = drand48() * (LIMITE_SUP - LIMITE_INF) + LIMITE_INF;
80        }
81
82      float aux[numeroHilos][4];
83
84      //Se inicia el cronometro de ciclos
85      start_counter();
86
87      //Inicio de la region paralela, en este punto se crean los hilos que se van
            a usar
88      #pragma omp parallel num_threads(numeroHilos) shared(aux)
89      {
90        //Se calcula el numero del hilo actual (se necesita este identificador)
91        int num = omp_get_thread_num();
92        float acumulacion[] = {0.0, 0.0, 0.0, 0.0}, mult[4];
93
94        //Se paraleliza el bucle for del computo
95        #pragma omp for
96        for(i = 0; i < total; i++)
97        {
98          //Se calcula la multiplicacion de a por b de la iteracion actual
99          mult[0] = *(a + i*4 + 0) * *(b + i*4 + 0) - *(a + i*4 + 1) * *(b + i*4
                + 1) - *(a + i*4 + 2) * *(b + i*4 + 2) - *(a + i*4 + 3) * *(b + i*4
```

```
            + 3);
100         mult[1] = *(a + i*4 + 0) * *(b + i*4 + 1) + *(a + i*4 + 1) * *(b + i*4
                + 0) + *(a + i*4 + 2) * *(b + i*4 + 3) - *(a + i*4 + 3) * *(b + i*4
                + 2);
101         mult[2] = *(a + i*4 + 0) * *(b + i*4 + 2) - *(a + i*4 + 1) * *(b + i*4
                + 3) + *(a + i*4 + 2) * *(b + i*4 + 0) + *(a + i*4 + 3) * *(b + i*4
                + 1);
102         mult[3] = *(a + i*4 + 0) * *(b + i*4 + 3) + *(a + i*4 + 1) * *(b + i*4
                + 2) - *(a + i*4 + 2) * *(b + i*4 + 1) + *(a + i*4 + 3) * *(b + i*4
                + 0);
103
104         //Una vez calculada se eleva al cuadrado y se acumula en el sumatorio
105         acumulacion[0] += mult[0] * mult[0] - mult[1] * mult[1] - mult[2] *
                mult[2] - mult[3] * mult[3];
106         acumulacion[1] += mult[0] * mult[1];
107         acumulacion[2] += mult[0] * mult[2];
108         acumulacion[3] += mult[0] * mult[3];
109       }
110
111       //Se indica a la matriz compartida que acumulaciones pertenecen a cada
              hilo usando el identificador
112       aux[num][0] = acumulacion[0];
113       aux[num][1] = acumulacion[1];
114       aux[num][2] = acumulacion[2];
115       aux[num][3] = acumulacion[3];
116     }
117
118     //Se suman las aportaciones de cada hilo en la variable final
119     for(i = 0; i < numeroHilos; i++)
120     {
121       reduccion[0] += aux[i][0];
122       reduccion[1] += aux[i][1];
123       reduccion[2] += aux[i][2];
124       reduccion[3] += aux[i][3];
125     }
126
127     //Se multiplica por 2 las filas pendientes (para ahorrar ciclos se extrajo
          factor comun de ambos sumatorios)
128     for(i = 1; i < 4; i++)
129       reduccion[i] *= 2;
130
131     //Se para el cronometro de ciclos
132     medidaCiclos = get_counter();
133
134     //Se muestra el resultado del sumatorio para ver si coincide con las otras
          versiones
135     printf("Resultado sumatorio: ");
136     representar_quaternion(reduccion);
137
138     //Se abre el fichero en modo append
139     if((fichero = fopen("registro4.txt", "a")) == NULL)
140     {
141       printf("Hubo un error al abrir el archivo en modo append\n");
142       return 1;
143     }
144
145     //Se escribe en el fichero el total de cuaterniones y la medida de ciclos
```

```
146     if(fprintf(fichero, "%d %f\n", total, medidaCiclos) < 0)
147     {
148       printf("Hubo un error al escribir el fichero\n");
149       return 1;
150     }
151 
152     //Se cierra el fichero
153     if(fclose(fichero) == EOF)
154     {
155       printf("Hubo un error al cerrar el archivo\n");
156       return 1;
157     }
158 
159     //Se libera la memoria de los vectores dinamicos utilizados
160     free(a);
161     free(b);
162 
163     //Finalizacion normal del programa
164     return 0;
165   }
166 
167   //Funcion para representar en pantalla un cuaternion
168   void representar_quaternion(float* quaternion)
169   {
170     printf("%f + %fi + %fj + %fk\n", *(quaternion + 0), *(quaternion + 1), *(
          quaternion + 2), *(quaternion + 3));
171   }
```

Código 14: compilación en terminal con OpenMP sin optimizaciones

```
icc -Wall -O0 version4.c -o version4 -qopenmp

./version1 <tamVector> <numeroHilos> <semillaAleatorios>
```

5.2. Resultados

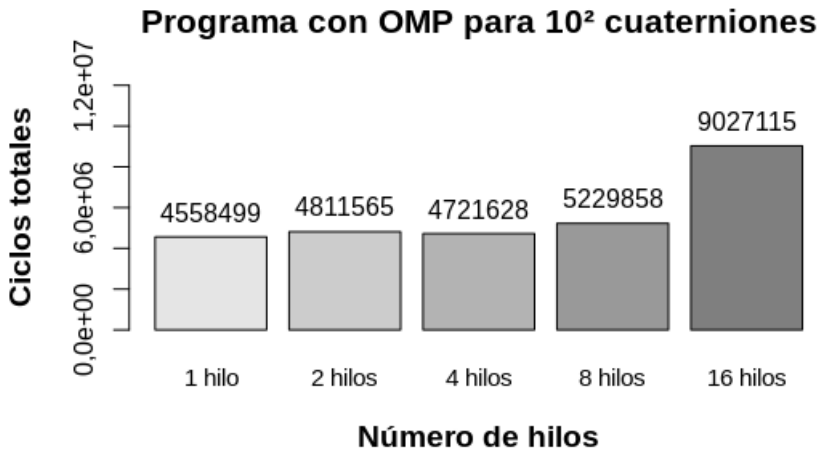

Figura 7: Resultados obtenidos con OMP para q = 2

El cómputo más pequeño muestra unos resultados pésimos para esta versión del código, especialmente cuando se usa una gran cantidad de hilos. El coste de la creación y gestión de hilos supera con creces al coste del propio cálculo.

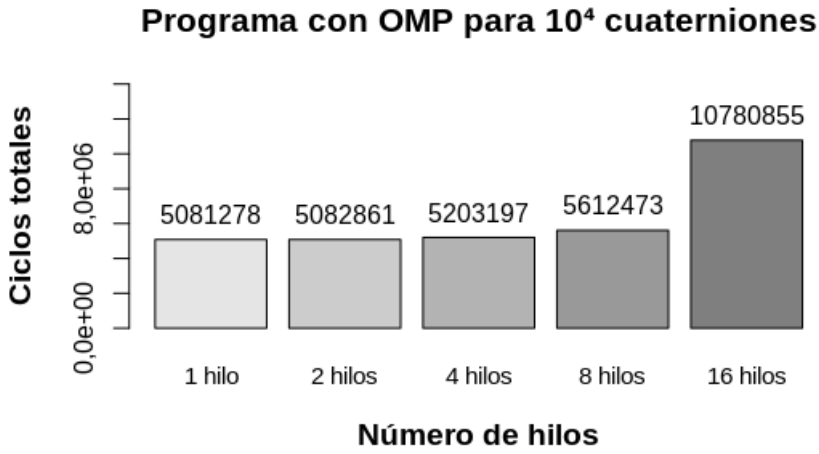

Figura 8: Resultados obtenidos con OMP para q = 4

Los resultados siguen sin mejorar debido a que la magnitud de la operación sigue siendo bastante pequeña y el coste de paralelizar el código sigue siendo muy elevado en comparación.

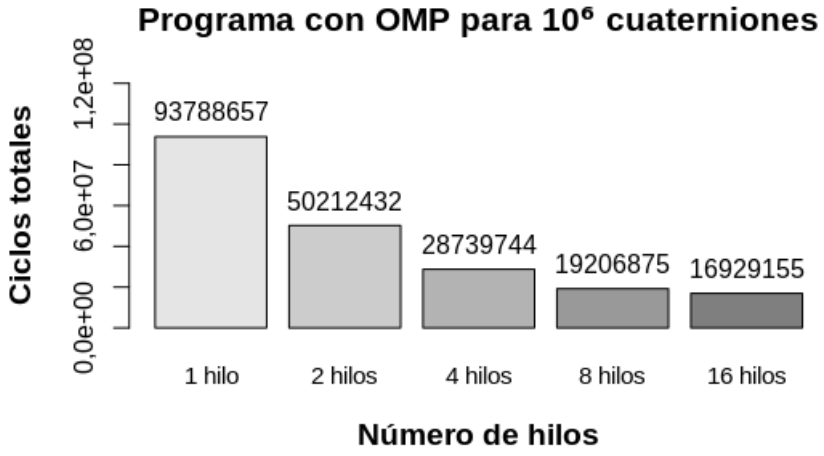

Figura 9: Resultados obtenidos con OMP para q = 6

.

Para una cantidad significativa de cuaterniones se empiezan a ver muy buenos resultados, sobretodo con muchos hilos. Para una operación tan costosa el coste y gestión de los hilos se vuelve muy pequeño en comparación al problema, se puede tolerar la penalización al conseguir enormes mejoras en los tiempos del cómputo.

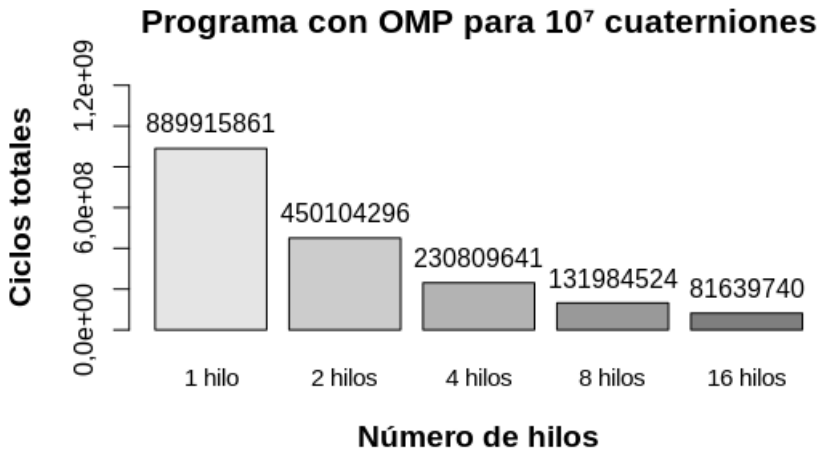

Figura 10: Resultados obtenidos con OMP para q = 7

.

Para la cantidad máxima exhibe los mejores resultados superando con mucho margen de diferencia al resto de versiones. Esto demuestra que cuando se tiene un cálculo muy costoso si se gestiona bien el uso de hilos se puede superar por mucho las optimizaciones del compilador. Es importante resaltar que la optimización del compilador usada en la versión 1 añade vectorización de datos, pero nunca entra en el terreno del cálculo en paralelo, por esa razón esta opción puede acabar superándolo.

6. Conclusiones

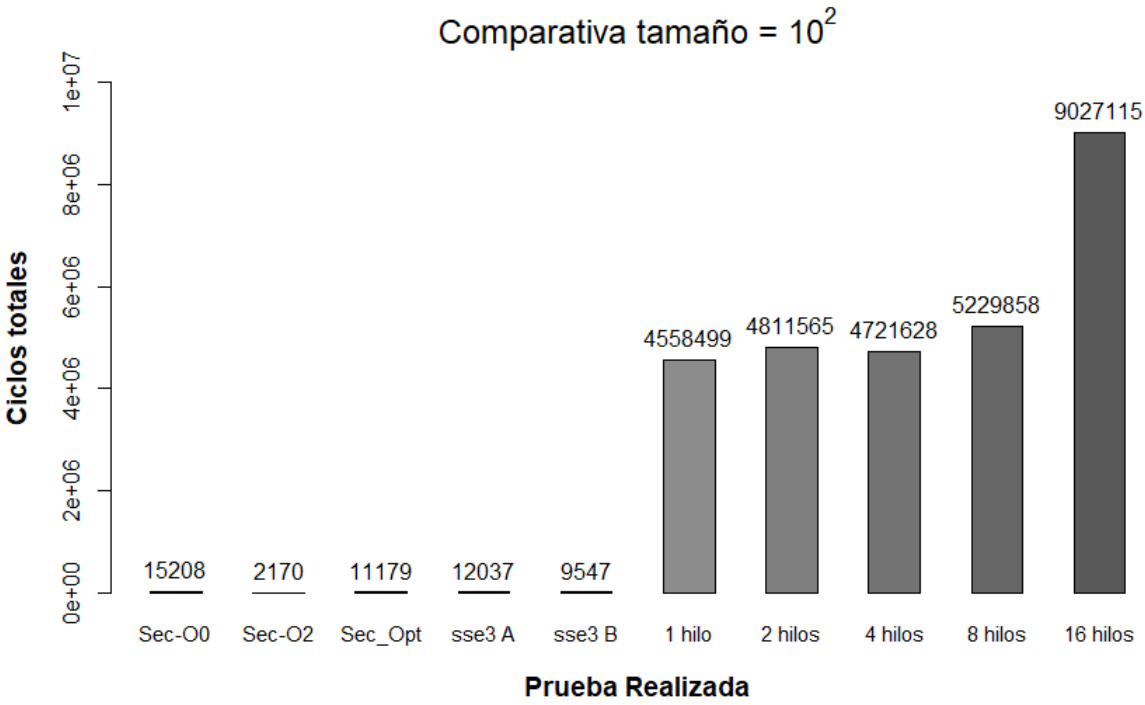

Figura 11: Resultados obtenidos al emplear 100 cuaterniones en las pruebas

Como se puede observar el mejor de los resultados, se obtiene cuando se compila el código secuencial base empleando el invel de optimización -O2, seguido de la vectorización por iteración de la multiplicación usando extensiones sse3.

Al usar omp el coste de creación y gestión de los hilos es bastante elevado, por lo que para una cantidad de cuaterniones tan pequeña, los resultados obtenidos con estos serán los peores y aumentará el número de ciclos a medida que se incrementen los hilos empleados.

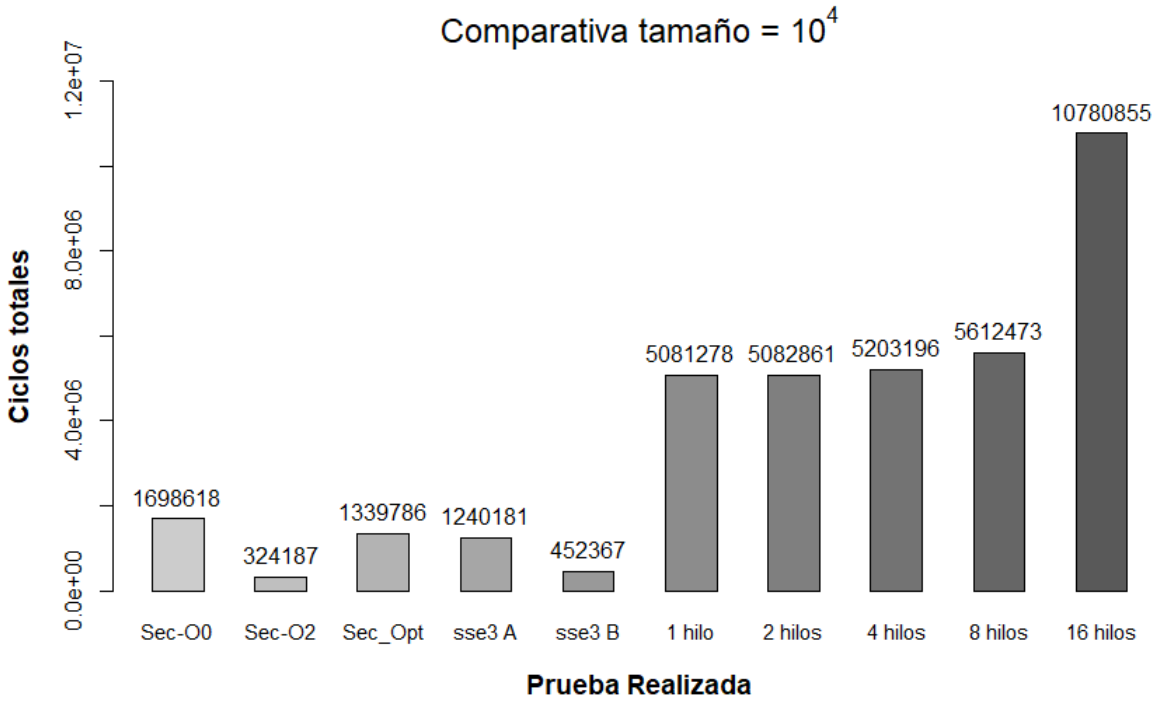

Figura 12: Resultados obtenidos al emplear 10000 cuaterniones en las pruebas

Al incrementarse el número, se sigue observando que las mejores opciones para estos números son la versión optimizada en -O2 del código secuencial y la vectorización de la iteraciones de la multiplicación, además de que el resto de versiones ajenas a omp empeoraron sus resultados considerablemente.

Las versiones con omp siguen demostrando que son ineficientes a la hora de tratar números tan pequeños, pues de hecho, se percibe un pequeño incremento en el número de ciclos.

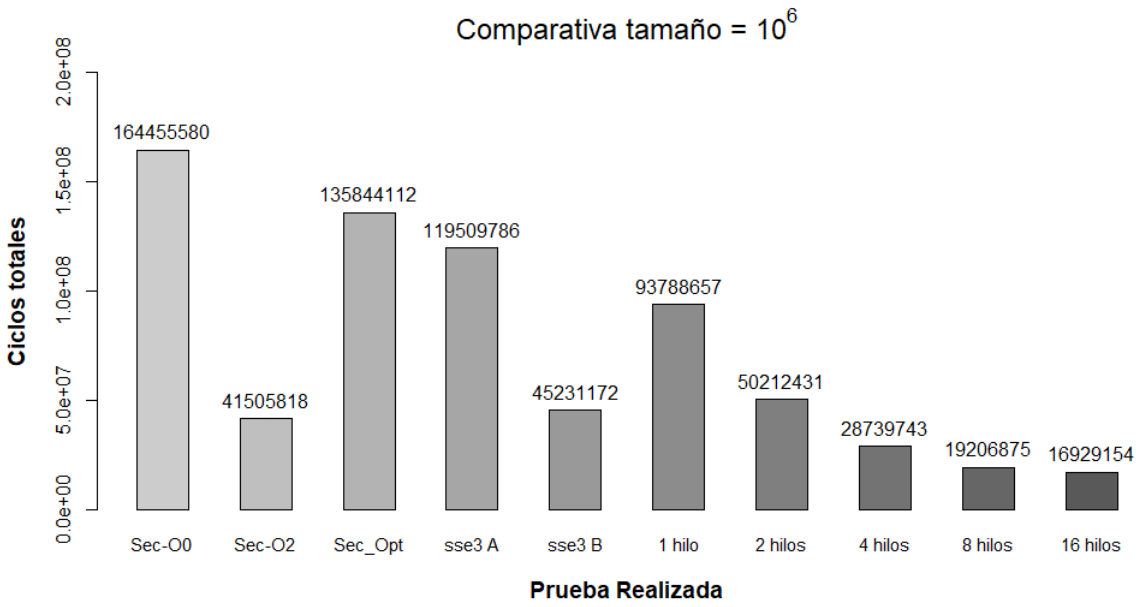

Figura 13: Resultados obtenidos al emplear 1000000 cuaterniones en las pruebas

Al trabajar con estos números, el uso de hilos comienza a mostrar resultados bastante buenos, puesto que cuando se trabaja con 4,8 y 16 hilos el número de ciclos es incluso inferior a la versión optimizada en -O2 y a la vectorización por iteración en see3. Además, cuando se trabaja con 1 y 2 hilos también se reduce el número de ciclos. Por otro lado, para el resto de versiones el número de ciclos sigue incrementándose con el número de cuaterniones a trabajar.

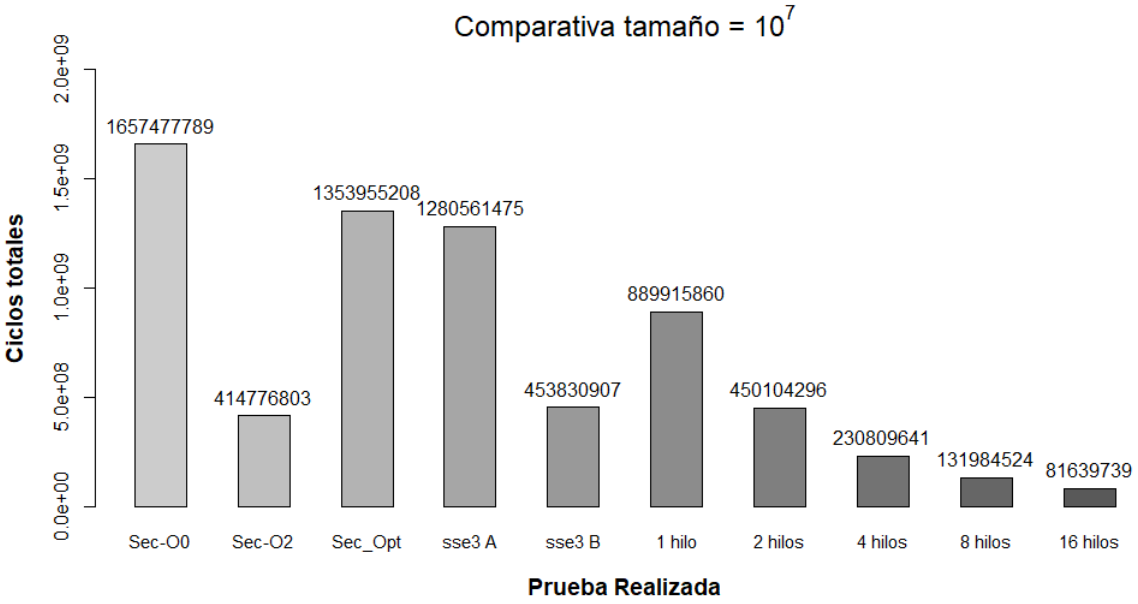

Figura 14: Resultados obtenidos al emplear 10000000 cuaterniones en las pruebas

Para el número máximo de cuaterniones, se obtiene una reducción drástica en el número de ciclos obtenidos en todas las versiones de omp, a excepción de cuando trabajamos con 1 hilo.

El resto de versiones siguen incrementando el número de ciclos que requieren, aunque la versión de omp con 2 hilos está bastante pareja a la versión optimizada en -O2 y la vectorización de las iteraciones en sse3.

Como conclusión, se puede decir que para números pequeños de cuaterniones es más rentable el uso de la versión secuencial base optimizada en -O2 y la vectorización de las iteraciones de sse3, puesto que no compensa el uso de hilos ya que se tiene que pagar un alto coste con el fin de gestionarlos y administrarlos. Esto provoca que se reduzca en gran medida la eficiencia del código para tamaños pequeños.

Por otro lado, para tamaños grandes el uso de hilos obtiene su coste mínimo pues en comparación al resto de versiones el coste computacional se reparte entre todos los hilos de forma que el número de ciclos se reduce considerablemente en comparación con el resto de versiones, cuyos ciclos no dejan de incrementarse a medida que aumenta el número de cuaterniones.

Referencias

[1] Procesador Intel® Xeon® E5-2650 v3 (caché de 25 M, 2,30 GHz) Especificaciones de productos, 2019. Intel [online]. [Citado el: 10 mayo 2019]. https://ark.intel.com/content/www/es/es/ark/products/81705/intel-xeon-processor-e5-2650-v3-25m-cache-2-30-ghz.html

[2] Intel Xeon E5-2650 v3 specifications, 2019. CPU-World [online]. [Citado el: 10 mayo 2019]. http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2650%20v3.html

[3] Quaternion, 2019. Wikipedia The Free Encyclopedia [online]. [Citado el: 17 abril 2019]. https://en.wikipedia.org/wiki/Quaternion

[4] BAKER, MARTIN J., [Sin fecha], Maths - Powers of Quaternions. Euclidean Space [online]. [Citado el: 18 abril 2019]. https://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/functions/power/index.htm

[5] Momchil Velikov, [Sin fecha], Fast SSE quaternion multiplication[online]. [Citado el: 20 abril 2019]. http://momchil-velikov.blogspot.com/2013/10/fast-sse-qutternion-multiplication.html