Programación Multinúcleo y extensiones SIMD

Rodríguez Carreira, Sergio

Tarrío Otero, Pablo

Índice.

Abstract. 1

Introducción. 1

Partes comunes. 2

Programa secuencial base. 3

Programa secuencial optimizado. 4

Vectorización de la multiplicación con AVX. 5

Vectorización por iteraciones del bucle con AVX. 7

OpenMP. 9

Análisis de resultados. 11

Conclusiones finales. 15

Anexo. 16

Bibliografía. 17

1. Abstract.

A través de la elaboración de una serie de códigos en lenguaje C, se caracteriza el coste temporal de las operaciones suma y multiplicación para los cuaterniones (computación del cuaternión), observando el efecto de las distintas implementaciones planteadas, así como de los diferentes tamaños del problema sobre el rendimiento del sistema, y realizando una interpretación de la misma.

2. Introducción.

En primer lugar, cabe destacar que todas las implementaciones programadas realizarán la misma tarea, que consistirá en la computación del cuaternión *dp*, el cuaternión resultante, a base de operar con dos vectores de cuaterniones (*a* y *b*), los datos iniciales. Para ello, primero se efectuará la multiplicación entre ambos vectores, cuaternión a cuaternión, y el resultado se almacenará en un vector auxiliar de cuaterniones (*c*). Después, se realizará una multiplicación del vector anteriormente calculado sobre sí mismo. Por último, el resultado de cada componente de cada cuaternión del vector tras el cálculo anterior se acumulará sobre el cuaternión resultante (que inicialmente fue inicializado con todo a 0s), se hará una adición entre cuaterniones, y al final de todas las iteraciones el cuaternión *dp* albergará la solución final.

```
Entrada:
   vectores de cuaterniones a(N),b(N);

Salida:
   cuaternion dp;

Computación:
   vector de cuaterniones c(N);
   dp=0; //inicialización de las componentes de dp a 0
   for i=1,N {
      c(i)=a(i)*b(i); //multiplicación de a por b
   }
   for i=1,N {
      dp=dp+c(i)*c(i); //multiplicación de c por sí mismo y
acumulación de resultados en dp
   }
```

Pseudocódigo de las operaciones a realizar.

Sin embargo, hay que tener muy en cuenta el tipo de dato con el que se trabaja, ya que en este caso no es muy usual: el cuaternión; el cual, básicamente, consiste en un conjunto (vector o estructura) de cuatro componentes (en cuanto a la programación, se considerarán de tipo *double*). Además, también hay que considerar que las operaciones entre cuaterniones, adición y producto, no son las mismas a las que se está acostumbrado [4]. En cuanto a la operación de adición, esta se realiza componente a componente, es decir, sumando componentes del mismo tipo de dos cuaterniones. En cuanto al producto, la operación ya cambia bastante, y sigue una determinada secuencia (véase la fórmula).

$$a + b = (a_1 + b_1) + (a_2 + b_2) + (a_3 + b_3) + (a_4 + b_4)$$

Adición.

$$ab = (a_1b_1-a_2b_2-a_3b_3-a_4b_4) + (a_1b_2+a_2b_1+a_3b_4-a_4b_3) + (a_1b_3-a_2b_4+a_3b_1+a_4b_2) + (a_1b_4+a_2b_3-a_3b_2+a_4b_1)$$

Producto.

Para la realización de las mediciones del coste temporal de computación, se utilizó un Lenovo Legion Y720-15IKB, cuyas características a resaltar son las siguientes:

- Microprocesador Intel ® Core (TM) i7-7700HQ con frecuencia de 2.80 GHz.
- 8 CPUs en línea, 2 hilos/núcleo, 4 núcleos/socket y 1 socket.
- Frecuencia de la CPU en un rango de 800-3800 MHz.

3. Partes comunes.

En este apartado se comentan una serie de aspectos generales que afectan a las programación de todas las diferentes implementaciones.

En primer lugar, incidir sobre la implementación de los cuaterniones, en este caso, este tipo de dato se construyó como una estructura formada por cuatro doubles (las cuatro componentes del cuaternión). Una vez ya se tiene esto, se puede proceder a definir los cuaterniones y vectores de cuaterniones necesarios para los programas.

En cuanto a los datos, se encuentran definidas tres constantes y una variable global. La primera es *CUAT*, la cual define el número de componentes que tiene un cuaternión, por lo que su valor será *4*. La segunda es *MAX_RAND*, generalmente de valor *1000*, y que marcará el rango máximo hasta el que se inicializan los valores de los componentes de los cuaterniones, para así no tener que trabajar con valores muy grandes. Y la tercera es *q*, cuyo valor se pide que sea uno de los siguientes: *2*, *4*, *6* y *7*; y que tendrá la función de hacer depender el valor del tamaño de los vectores (una variable global entera: *N*) de la siguiente manera: $N = 10^q$, por lo que esta constante definirá el tamaño del problema.

Además, será necesario realizar, antes de proceder con las operaciones, todas las inicializaciones necesarias. Como se comentó anteriormente, las componentes de los cuaterniones de los vectores *a* y *b* se inicializan con valores aleatorios, acotados por el valor de *MAX_RAND*. También será necesario inicializar cada componente del cuaternión resultante *dp* a 0, puesto que sobre él se van a ir acumulando los resultados de cada operación, por lo que se deberá partir de un valor nulo.

```
void inicializacion(cuaternions *a, cuaternions *b) {
   srand(1); // Misma semilla de aleatorios para cada programa
   for (unsigned long int i = 0; i < N; i++) {
     // Asignacion de valor aleatorio entre 0 y MAX_RAND por componente
     a[i].c1 = rand() % MAX_RAND, a[i].c2 = rand() % MAX_RAND;
     a[i].c3 = rand() % MAX_RAND, a[i].c4 = rand() % MAX_RAND;
     b[i].c1 = rand() % MAX_RAND, b[i].c2 = rand() % MAX_RAND;
     b[i].c3 = rand() % MAX_RAND, b[i].c4 = rand() % MAX_RAND;
   }
}
dp.c1 = 0; dp.c2 = 0; dp.c3 = 0; dp.c4 = 0;
```

Inicialización de los datos.

Después de toda la inicialización, comenzaría la ejecución de las diferentes operaciones que se requieren en para la resolución de la práctica, con implementaciones diferentes para cada apartado, junto a las mediciones de tiempo de las mismas.

Cabe destacar que las mediciones realizadas en cada uno de los apartados serán en cuanto al número de ciclos por cada cuaternión, es decir, se medirá el número de ciclos totales que lleva realizar todas las operaciones, y se dividirá el valor entre el número total de cuaterniones calculados: *N*. De esta manera se obtiene una forma de medir el tiempo que tardará en realizarse la operación en cada cuaternión. Por último, añadir que los valores tomados de los ciclos se basan en la mediana de 10 mediciones diferentes de los tiempos de ejecución de cada código, usando la opción del sistema *perf stat -d* para un uso de la frecuencia óptimo. Para obtener la mediana, se realiza la media de los dos datos centrales, pues el número de datos siempre será par, y no habrá un valor central único.

Los resultados de las operaciones para los distintos valores de *N* son los siguientes:

	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
dp1	-93200530252571.00	-8919457025719845.00	-886280055946355584.00	-8861404163789755392.00
dp2	-52059569815002.00	-4997792671911976.00	-497854825251262016.00	-4974158169049270272.00
dp3	-56885340495126.00	-4880690141947416.00	-496785054328547328.00	-4975996571538761728.00
dp4	-61026245604180.00	-4983202507575840.00	-497380455255206656.00	-4979621342586621952.00

No obstante, hay que tener en cuenta que este resultado podría variar ligeramente en alguna de las implementaciones, esto se debe al tipo de operaciones que se realiza en cada uno y a los valores con los que se operan, pudiendo causar problemas si son altos (de ahí el motivo del uso de *MAX_RAND*).

4. Programa secuencial base.

En el programa secuencial base se modularizan dos funciones, una para cada tipo de operación: producto y adición. La primera de ellas, realiza el producto de dos vectores de cuaterniones, cuaternión a cuaternión, mediante un bucle *for*, y con operaciones básicas (la suma y multiplicación tradicionales), y se usará tanto para la multiplicación de a por *b* (resultado *c*) como para la de *c* por *c*. La segunda de ellas, realiza la suma de el vector de cuaterniones *c con dp*, acumulando cada resultado, cuaternión a cuaternión, sobre *dp*. Por lo tanto, se estarían realizando un total de 3 bucles con *N* iteraciones: dos por las dos multiplicaciones y otro por la adición. Por lo que el código de esta implementación queda de la siguiente forma:

```
// (código)... inicializaciones y comienzo de medición de tiempos
   c = multiplicacion(a, b); // c = a*b
   adicion(&dp, multiplicacion(c, c)); // dp = dp + c*c
// (código)... fin de medición de tiempos y se muestran los resultados

/* Multiplicación de dos vectores de cuaterniones: cuaternion a cuaternion */
cuaternions *multiplicacion(cuaternions *a, cuaternions *b) {
```

```
    cuaternions *r = malloc(N * sizeof(cuaternions)); // resultado
    for (unsigned long int i = 0; i < N; i++) {
        // A cada iteración se realiza el producto de dos cuaterniones, de la misma
posición, dando como resultado otro cuaternión
        r[i].c1 = a[i].c1*b[i].c1-a[i].c2*b[i].c2-a[i].c3*b[i].c3-a[i].c4*b[i].c4;
        r[i].c2 = a[i].c1*b[i].c2+a[i].c2*b[i].c1+a[i].c3*b[i].c4-a[i].c4*b[i].c3;
        r[i].c3 = a[i].c1*b[i].c3-a[i].c2*b[i].c4+a[i].c3*b[i].c1+a[i].c4*b[i].c2;
        r[i].c4 = a[i].c1*b[i].c4+a[i].c2*b[i].c3-a[i].c3*b[i].c2+a[i].c4*b[i].c1;
    }
    return r;
}

/* Suma de un vector de cuaterniones: acumulación del valor de cada componente */
void adicion(cuaternions *dp, cuaternions *a) {
    for (unsigned long int i = 0; i < N; i++) {
        dp->c1 += a[i].c1, dp->c2 += a[i].c2, dp->c3 += a[i].c3, dp->c4 += a[i].c4;
    }
}
```

*Fragmento de código del ejercicio 1.

Además, este código (una única implementación) debe ser compilado de dos formas diferentes. Por una lado, se pide que se haga sin optimizaciones del compilador, por lo que habrá que colocar la siguiente *flag* de opciones cuando se vaya a compilar el código: `-O0`; de esta forma se indica que no se realice ningún tipo de optimización sobre el código, disminuyendo el tiempo de compilación. En este caso, la medición de tiempos, en función del valor de *N*, será la siguiente:

	100 (q = 2)	10000 (q = 4)	1000000 (q = 6)	10000000 (q = 7)
ciclos / cuaternión	173.43	163.727	161.623	160.366

Por el otro lado, se pide hacerlo con optimizaciones del compilador, incluída la autovectorización, por lo que habrá realizar la compilación con las siguiente *flags*: `-mavx -O3`. Para este caso, los tiempos obtenidos, en función del valor de *N*, son los siguientes:

	100 (q = 2)	10000 (q = 4)	1000000 (q = 6)	10000000 (q = 7)
ciclos / cuaternión	84.58	76.535	76.264	77.276

5. Programa secuencial optimizado.

Esta implementación tiene la misma mecánica que el apartado anterior, pero con la diferencia de que todas las operaciones se realizan en un único bucle *for*. De esta forma, a cada iteración, se realizará en primer lugar la operación de multiplicación de un cuaternión del vector *a* con uno del vector *b*, el resultado de esta se guarda en un cuaternión auxiliar *c*, que se usará, justo después, para realizar la multiplicación consigo mismo, y acumular el resultado de cada componente sobre el cuaternión final *dp*. En cuanto al código, se realiza de la siguiente forma:

```
// (código)... inicializaciones y comienzo de medición de tiempos
   operacion(&dp, a, b); // c = a*b  y  dp = dp + c*c
// (código)... fin de medición de tiempos y se muestran los resultados

/* Se realizan todas las operaciones (productos y adición) con los dos vectores de
cuaterniones en un único bucle */
void operacion(cuaternions *dp, cuaternions *a, cuaternions *b) {
   cuaternions c; // Cuaternion auxiliar
   for (unsigned long int i = 0; i < N; i++) {
      // Producto de los dos vectores iniciales: a*b, cuaternion a cuaternion
      c.c1 = a[i].c1*b[i].c1-a[i].c2*b[i].c2-a[i].c3*b[i].c3-a[i].c4*b[i].c4;
      c.c2 = a[i].c1*b[i].c2+a[i].c2*b[i].c1+a[i].c3*b[i].c4-a[i].c4*b[i].c3;
      c.c3 = a[i].c1*b[i].c3-a[i].c2*b[i].c4+a[i].c3*b[i].c1+a[i].c4*b[i].c2;
      c.c4 = a[i].c1*b[i].c4+a[i].c2*b[i].c3-a[i].c3 * b[i].c2+a[i].c4*b[i].c1;
      // Multiplicación de c sobre sí mismo y acumulación sobre el resultado
      dp->c1 += c.c1 * c.c1 - c.c2 * c.c2 - c.c3 * c.c3 - c.c4 * c.c4;
      dp->c2 += c.c1 * c.c2 + c.c2 * c.c1 + c.c3 * c.c4 - c.c4 * c.c3;
      dp->c3 += c.c1 * c.c3 - c.c2 * c.c4 + c.c3 * c.c1 + c.c4 * c.c2;
      dp->c4 += c.c1 * c.c4 + c.c2 * c.c3 - c.c3 * c.c2 + c.c4 * c.c1;
   }
}
```

*Fragmento de código, para mayor contenido véase archivo: *ej2.c*

Para esta implementación, habrá que realizar la compilación de una única forma, mediante la *flag*: `-O0`; indicando que no se requieren optimizaciones por parte del compilador. Los tiempos obtenidos son los siguientes:

	100 (q = 2)	10000 (q = 4)	1000000 (q = 6)	10000000 (q = 7)
ciclos / cuaternión	66.87	67.144	69.102	64.970

6. Vectorización de la multiplicación con AVX.

En el primer caso de la implementación del código con extensiones de AVX, se pide la vectorización de la operación de multiplicación de los cuaterniones. Para ello, se necesitarán una serie de variables del tipo *__m256d*, que básicamente consiste en un vector de cuatro *doubles* que permite trabajar con las funciones de AVX, así como un vector *c* del tipo *double*, también de cuatro componentes, que se utilizará para realizar conversiones entre ambos tipos de vectores. Además, una de esas variables del tipo *__m256d* deberá ser inicializada con sus componentes a 0, ya que actuará como *dp* auxiliar.

Al comienzo, por cada iteración, será necesario asignar las componentes de cada cuaternión de *a* y *b* a una variable *__m256d*, mediante la función *_mm256_set_pd*, la cual recibe como parámetros 4 doubles. Hay que tener en cuenta que el orden de asignación de los componentes con esta función es el inverso.

Tras dicha inicialización, comienzan las operaciones. En primer lugar, se realiza el producto de *a* por *b*; cabe destacar que en este apartado la multiplicación se realizará por columnas (4 registros), es decir, cada vector contendrá una porción de la multiplicación de cada componente, por lo que al final habrá que sumar todos esos vectores. Para conseguir

el orden adecuado de la componentes de los cuaterniones para poder realizar las operaciones, se utilizarán una serie de shuffles con la función *_mm256_permute4x64_pd*, la cual recibe como parámetros el vector *__m256d* sobre el que se quiere realizar el shuffle, y la máscara (representada en decimal, no en binario) que decide cómo se realizará la mezcla.

Una vez se ordenaron correctamente las componentes de los vectores, se procede a la multiplicación de componentes de un cuaternión de *a* con otro de *b*, mediante la función *_mm256_mul_pd*, de esta forma, se obtendrán cuatro columnas, las cuales se deberán sumar, con la función *_mm256_add_pd* para obtener así el cuaternión resultante de la multiplicación (*c*). Cabe destacar que algunas partes de la multiplicación requieren de cambios de signo, y esto se consigue mediante el uso de una variable que tiene asignados los signos correspondientes.

Por último, se deberá realizar de nuevo la multiplicación del vector resultante consigo mismo, que se hará de la misma manera que la multiplicación antes mencionada. Sin embargo, ahora la suma de las diversas columnas se acumulará, de forma vectorizada, sobre el vector auxiliar *dp*. Tras realizar todas las iteraciones, y tener el valor final de *dp*, será necesario convertirlo de un tipo de dato *__m256d*, a un vector normal de 4 doubles, mediante la función *_mm256_store_pd*, y cada componente del vector se igualará a las componentes de la estructura de *dp*, para obtener así el resultado final.

```
// (código)... inicializaciones y comienzo de medición de tiempos
   operacion(&dp, a, b); // c = a*b  y  dp = dp + c*c
// (código)... fin de medición de tiempos y se muestran los resultados

/* Se realizan todas las operaciones (productos y adición) de los dos vectores en un
único bucle, con procesamiento vectorial, vect. la multiplicación de cuats. */
void operacion(cuaternions *dp, cuaternions *a, cuaternions *b) {
   double *c = _mm_malloc(sizeof(cuaternions), 32);
   __m256d aux_a, aux_b, aux_c;
   __m256d columna1, columna2, columna3, columna4; // Valores columnas mult.
   __m256d signos, dp_aux = _mm256_set_pd(0, 0, 0, 0);

   // Las operaciones se realizan por columnas en base a la multiplicación
   for (int i = 0; i < N; i++) {
      aux_a = _mm256_set_pd(a[i].c4, a[i].c3, a[i].c2, a[i].c1); // Datos iniciales
      aux_b = _mm256_set_pd(b[i].c4, b[i].c3, b[i].c2, b[i].c1);

      /* MULTIPLICACIÓN a*b */
      columna1 = _mm256_mul_pd(_mm256_permute4x64_pd(aux_a,0),aux_b);

      signos = _mm256_set_pd(1, -1, 1, -1);
      columna2 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_a, 85),
_mm256_permute4x64_pd(aux_b, 177)));

      signos = _mm256_set_pd(-1, 1, 1, -1);
      columna3 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_a, 170),
_mm256_permute4x64_pd(aux_b, 78)));

      signos = _mm256_set_pd(1, 1, -1, -1);
      columna4 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_a, 255),
_mm256_permute4x64_pd(aux_b, 27)));

      // Suma de las cuatro columnas, resultado en c, correspondiendo a la mult. a*b
      aux_c = _mm256_add_pd(_mm256_add_pd(columna1, columna2), _mm256_add_pd(columna3,
columna4));
```

```
      /* MULTIPLICACION c*c */
      columna1 = _mm256_mul_pd(_mm256_permute4x64_pd(aux_c, 0), aux_c);

      signos = _mm256_set_pd(1, -1, 1, -1);
      columna2 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_c, 85),
_mm256_permute4x64_pd(aux_c, 177)));

      signos = _mm256_set_pd(-1, 1, 1, -1);
      columna3 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_c, 170),
_mm256_permute4x64_pd(aux_c, 78)));

      signos = _mm256_set_pd(1, 1, -1, -1);
      columna4 = _mm256_mul_pd(signos, _mm256_mul_pd(_mm256_permute4x64_pd(aux_c, 255),
_mm256_permute4x64_pd(aux_c, 27)));

      /* ADICIÓN a dp */
      // Suma 4 columnas, y el resultado se acumula sobre lo que se tenia: dp+=c*c
      dp_aux = _mm256_add_pd(dp_aux, _mm256_add_pd(_mm256_add_pd(columna1, columna2),
_mm256_add_pd(columna3, columna4)));
   }
   _mm256_store_pd(c, dp_aux); // Se pasa de un vector del tipo AVX a uno normal
   dp->c1 = c[0], dp->c2 = c[1], dp->c3 = c[2], dp->c4 = c[3]; // Asignación dp
   _mm_free(c); // Liberación de memoria
}
```

*Fragmento de código, para mayor contenido véase archivo: *ej3a.c*

En este primer caso de la implementación con extensiones de AVX, debido a que se utilizan funciones de AVX2 (como *_mm256_permute4x64_pd*), se deberá compilar el código con las siguientes *flags*: `-m32 -mavx2 -O0`; como se puede observar, se excluyen las optimizaciones del compilador. Los tiempos obtenidos son los siguientes:

	100 (q = 2)	10000 (q = 4)	1000000 (q = 6)	10000000 (q = 7)
ciclos / cuaternión	130.29	114.442	116.552	106.971

7. Vectorización por iteraciones del bucle con AVX.

En el segundo caso de la implementación del código con extensiones de AVX, se pide vectorizar por iteraciones del bucle. Para ello, al igual que en el primer caso de AVX, se deben definir una serie de variables del tipo *__m256d*, así como el vector de cuatro *doubles*, *c*, o la inicialización del *dp_aux* con sus componentes a 0.

Para vectorizar por iteraciones del bucle, se irán realizando de forma simultánea la multiplicación de cuatro pares de cuaterniones de los vectores *a* y *b*. Para ello, primero se almacenarán las componentes de los cuaterniones en una serie de vectores del tipo *__m256d* agrupando en una misma variable componentes del mismo vector y con el mismo componente (índice) de los cuatro cuaterniones, p.e.: almacenar en el mismo vector la primera componente de los cuatro cuaterniones de *a* a los que se acceden. Esta asignación se realiza mediante la función del *_mm256_set_pd*, en la que se especifican los cuatro *doubles* que se van a almacenar en el vector.

A continuación, se deberá vectorizar la multiplicación de los cuaterniones. Para ello, se hará uso de las siguientes funciones: *_mm256_fmadd_pd* y *_mm256_fmsub_pd*, según sea conveniente. Estas funciones se encargan de realizar la multiplicación de dos vectores *__m256d* y sumar/restar el resultado con otro vector del mismo tipo (cabe destacar que hay que tener cuidado con los signos al usar la función de la resta). De esta forma, se realiza el producto de cada componente de un cuaternión con el otro, resultando en otros cuatro vectores (equivalente a cuatro vectores *c*), cada uno con componentes del mismo índice.

A continuación, se deberá volver a realizar la multiplicación sobre las variables que resultaron de la operación anterior consigo mismas, y a continuación, se vectoriza la suma de dichas componentes, acumulando el valor sobre el que se tenía de *dp_aux*. Para poder realizar la suma, se recurre a las funciones *_mm256_hadd_pdd*, *_mm256_permute4x64_pd* y *_mm256_add_pd*. Tras la acumulación de las componentes mediante la primera de las funciones indicadas, y un mezclado de este, se consigue obtener una combinación correcta para acumular los datos en la variable auxiliar *dp_aux*. Una vez terminaron la iteraciones, se pasa el resultado al cuaternión *dp*, como en la anterior implementación.

```
// (código)... inicializaciones y comienzo de medición de tiempos
   operacion(&dp, a, b); // c = a*b  y  dp = dp + c*c
// (código)... fin de medición de tiempos y se muestran los resultados

/* Se realizan todas las operaciones (productos y adición) de los dos vectores en un
único bucle, con procesamiento vectorial, vect. por iteraciones del bucle */
void operacion(cuaternions *dp, cuaternions *a, cuaternions *b) {
   double *c = _mm_malloc(4 * CUAT * sizeof(double), 32);

   __m256d a1, a2, a3, a4, b1, b2, b3, b4; // Almacena componentes de los cuaterniones
   __m256d com1, com2, com3, com4; // Almacena valores tras 1° mult.
   __m256d c1, c2, c3, c4; // Almacena valores de la mult. c*c
   __m256d dp_aux = _mm256_set_pd(0, 0, 0, 0);

   for (int i = 0; i < N; i += 4) { // Operaciones con 4 cuaterniones simultaneos
      // En cada vector se guardan componentes del mismo tipo y del mismo vector
      a1 = _mm256_set_pd(a[i + 3].c1, a[i + 2].c1, a[i + 1].c1, a[i].c1);
      a2 = _mm256_set_pd(a[i + 3].c2, a[i + 2].c2, a[i + 1].c2, a[i].c2);
      a3 = _mm256_set_pd(a[i + 3].c3, a[i + 2].c3, a[i + 1].c3, a[i].c3);
      a4 = _mm256_set_pd(a[i + 3].c4, a[i + 2].c4, a[i + 1].c4, a[i].c4);
      b1 = _mm256_set_pd(b[i + 3].c1, b[i + 2].c1, b[i + 1].c1, b[i].c1);
      b2 = _mm256_set_pd(b[i + 3].c2, b[i + 2].c2, b[i + 1].c2, b[i].c2);
      b3 = _mm256_set_pd(b[i + 3].c3, b[i + 2].c3, b[i + 1].c3, b[i].c3);
      b4 = _mm256_set_pd(b[i + 3].c4, b[i + 2].c4, b[i + 1].c4, b[i].c4);

      /* MULTIPLICACIÓN a*b */
      // Producto entre componentes de cuaterniones, resultando en 4 vectores 'c'
      com1 = _mm256_fmsub_pd(a1, b1, _mm256_fmadd_pd(a2, b2, _mm256_fmadd_pd(a3, b3,
_mm256_mul_pd(a4, b4))));
      com2 = _mm256_fmadd_pd(a1, b2, _mm256_fmadd_pd(a2, b1, _mm256_fmsub_pd(a3, b4,
_mm256_mul_pd(a4, b3))));
      com3 = _mm256_fmadd_pd(a1, b3, _mm256_fmadd_pd(a3, b1, _mm256_fmsub_pd(a4, b2,
_mm256_mul_pd(a2, b4))));
      com4 = _mm256_fmadd_pd(a1, b4, _mm256_fmsub_pd(a2, b3, _mm256_fmsub_pd(a3, b2,
_mm256_mul_pd(a4, b1))));

      /* MULTIPLICACIÓN c*c */
      c1 = _mm256_fmsub_pd(com1, com1, _mm256_fmadd_pd(com2, com2, _mm256_fmadd_pd(com3,
com3, _mm256_mul_pd(com4, com4))));
      c2 = _mm256_fmadd_pd(com1, com2, _mm256_fmadd_pd(com2, com1, _mm256_fmsub_pd(com3,
com4, _mm256_mul_pd(com4, com3))));
```

```
      c3 = _mm256_fmadd_pd(com1, com3, _mm256_fmadd_pd(com3, com1, _mm256_fmsub_pd(com4,
com2, _mm256_mul_pd(com2, com4))));
      c4 = _mm256_fmadd_pd(com1, com4, _mm256_fmsub_pd(com2, com3, _mm256_fmsub_pd(com3,
com2, _mm256_mul_pd(com4, com1))));

      // ADICIÓN a dp: Suma componentes mismo tipo, y acumulacion resultado: dp+=c*c
      dp_aux = _mm256_add_pd(dp_aux,
_mm256_hadd_pd(_mm256_permute4x64_pd(_mm256_hadd_pd(c1, c3), 216),
_mm256_permute4x64_pd(_mm256_hadd_pd(c2, c4), 216)));
   }
   _mm256_store_pd(c, dp_aux); // Se pasa de un vector del tipo AVX a uno normal
   dp->c1 = c[0], dp->c2 = c[1], dp->c3 = c[2], dp->c4 = c[3]; // Asignación dp
   _mm_free(c); // Liberación de memoria
}
```

*Fragmento de código, para mayor contenido véase archivo: *ej3b.c*

En este segundo caso de la implementación con extensiones de AVX, también se utilizan funciones de AVX2, al utilizar las permutaciones. No obstante, el uso de las operaciones de *_mm256_fmadd_pd* y *_mm256_fmsub_pd*, hacen que se deba incluir una *flag* más, quedando de la siguiente forma: `-m32 -mavx2 -mfma -O0`; además de que no tiene optimizaciones del compilador. Los tiempos obtenidos son los siguientes:

	100 (q = 2)	10000 (q = 4)	1000000 (q = 6)	10000000 (q = 7)
ciclos / cuaternión	106.22	59.068	62.582	55.018

8. OpenMP.

Para esta última implementación, se pide realizar un código en el que se implemente la computación del cuaternión por medio de OpenMP. En este caso, habrá que declarar una constante adicional: *k*, la cual indica el número de hilos que se utilizarán en la paralelización del fragmento de código de la funciones. Puesto que el ordenador utilizado para la toma de mediciones consta de 8 núcleos, se realizaron los siguientes casos de prueba en función del valor de *k*: *1, 2, 4, 6* y *8*.

Tras el inicio del contador para la medición de tiempos, se declara un `#pragma omp parallel`, el cual indica que el fragmento de código posterior se va a paralelizar entre una serie de hilos. Por lo tanto, se deberán hacer una serie de indicaciones en esa declaración: el número de hilos que se van a usar (*k*) y una serie de variables privadas para cada hilo (*i*: utilizado como índice para las iteraciones del bucle, *auxdp*: tratado como una variable auxiliar de *dp* que sirve para guardar las sumas parciales que realice cada hilo, y *c*: un cuaternión auxiliar usado para la multiplicación).

Tras declarar la paralelización de la operación de los cuaterniones, se deberán inicializar los datos privados de cada hilo, en este caso sólo será necesario iniciar *auxdp* con sus componentes a 0, para acumular la suma parcial que calcule cada hilo por separado, puesto que se acumulará más adelante sobre la variable definitiva.

A continuación, comienzan las operaciones. En primer lugar se declara un `#pragma omp for`, el cual indica que en el bucle siguiente que se declara, sus iteraciones se van a repartir entre los *k* hilos indicados. En dicho bucle, a cada iteración, se realizará la operación de multiplicación de un cuaternión del vector *a* con uno del vector *b*, el resultado de esta se guarda en un cuaternión auxiliar *c*, que se usará, justo después, para realizar la multiplicación consigo mismo, y acumular el resultado de cada componente sobre un cuaternión parcial *auxdp*.

Por último, se tendrá que realizar la adición de cada una de las sumas parciales que calculó cada hilo sobre la computación del cuaternión en *auxdp*. Para ello, cada adición de cada componente de *auxdp* sobre uno de *dp* se realizará en un `#pragma omp atomic`, de esta forma se asegura que solo un hilo de forma simultánea realizará la adición de su suma parcial a la total, y así se evita que varios hilos puedan sumar sus resultados a la vez, lo cual produciría una condición de carrera con un resultado erróneo del cuaternión final *dp*.

```
// Código que se paraleliza para 'k' hilos con unas variables privadas cada uno
#pragma omp parallel private(i, auxdp, c) num_threads(k) {
   auxdp.c1 = 0, auxdp.c2 = 0, auxdp.c3 = 0, auxdp.c4 = 0;
   // Las iteraciones de este bucle se reparten entre los diferentes hilos
   #pragma omp for
   for (i = 0; i < N; i++) {
      // Cada iteración implica la multiplicación de un cuaternión de a y b
      c.c1 = a[i].c1*b[i].c1-a[i].c2*b[i].c2-a[i].c3*b[i].c3-a[i].c4*b[i].c4;
      c.c2 = a[i].c1*b[i].c2+a[i].c2*b[i].c1+a[i].c3*b[i].c4-a[i].c4*b[i].c3;
      c.c3 = a[i].c1*b[i].c3-a[i].c2*b[i].c4+a[i].c3*b[i].c1+a[i].c4*b[i].c2;
      c.c4 = a[i].c1*b[i].c4+a[i].c2*b[i].c3-a[i].c3*b[i].c2+a[i].c4*b[i].c1;
      // El resultado se multiplica consigo mismo y se acumula de forma parcial
      auxdp.c1 += c.c1 * c.c1 - c.c2 * c.c2 - c.c3 * c.c3 - c.c4 * c.c4;
      auxdp.c2 += c.c1 * c.c2 + c.c2 * c.c1 + c.c3 * c.c4 - c.c4 * c.c3;
      auxdp.c3 += c.c1 * c.c3 - c.c2 * c.c4 + c.c3 * c.c1 + c.c4 * c.c2;
      auxdp.c4 += c.c1 * c.c4 + c.c2 * c.c3 - c.c3 * c.c2 + c.c4 * c.c1;
   }
   // Solo un hilo simultáneamente añade su suma parcial a la total
   #pragma omp atomic
      dp.c1 += auxdp.c1;
   #pragma omp atomic
      dp.c2 += auxdp.c2;
   #pragma omp atomic
      dp.c3 += auxdp.c3;
   #pragma omp atomic
      dp.c4 += auxdp.c4;
}
```

*Fragmento de código, para mayor contenido véase archivo: *ej4.c*

Para cada una de las cinco ejecuciones que se realizan para la implementación con OpenMP (una ejecución por cada valor del número de hilos *k*), tampoco se utilizan optimizaciones del compilador. Por lo tanto, se requieren utilizar las siguientes *flags*: `-O0 -fopenmp`. Y los resultados de las medidas temporales son:

k = 1	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
ciclos / cuaternión	217.04	69.395	66.043	66.058

k = 2	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
ciclos / cuaternión	1424.83	53.076	36.306	35.068

k = 4	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
ciclos / cuaternión	3430.73	67.542	20.380	18.691

k = 6	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
ciclos / cuaternión	4668.13	75.288	24.248	23.310

k = 8	100 (*q* = 2)	10000 (*q* = 4)	1000000 (*q* = 6)	10000000 (*q* = 7)
ciclos / cuaternión	14692.56	232.085	22.109	18.207

9. Análisis de resultados.

Para las conclusiones de la práctica propuesta, se analizarán los datos para los distintos valores de *q* existentes y las diversas implementaciones planteadas. Para la representación, se utilizarán gráficos de barras, de esta forma se podrá observar las medidas de los ciclos por cuaternión calculado para cada implementación en cada uno de los diferentes valores de *q*. En el eje X se indicará el tipo de implementación, mientras que el eje Y será utilizado para representar los ciclos por cuaternión.

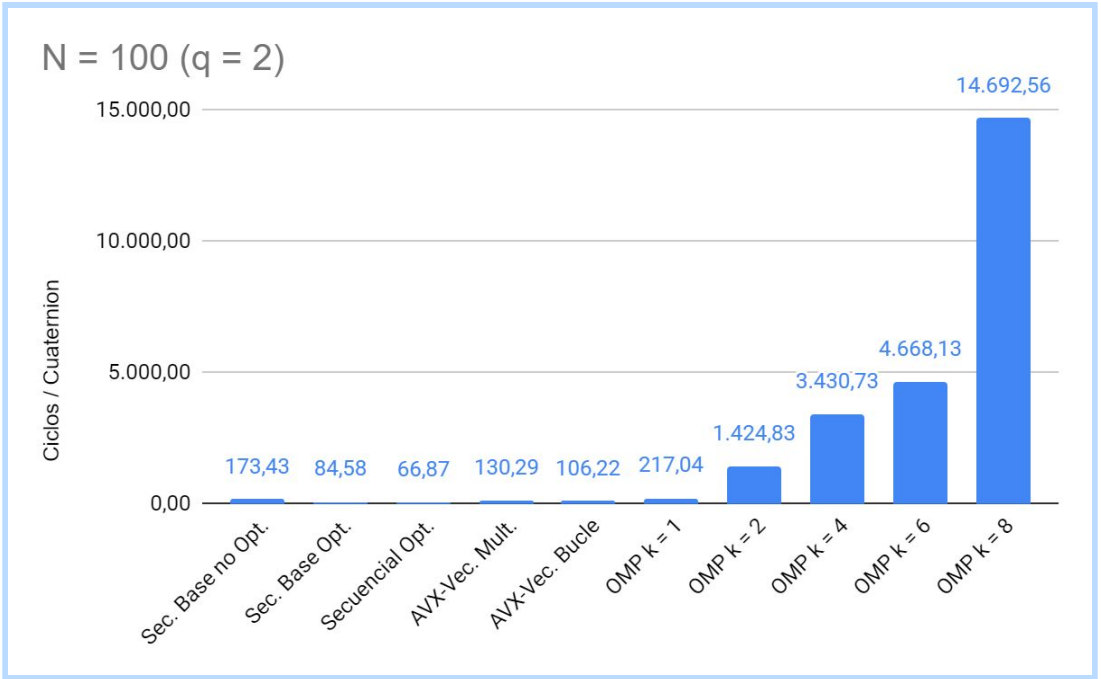

Figura 1. ciclos/cuaternión para cada implementación con *q*=2.

En el caso de *q=2*, se puede observar una gran diferencia entre las diversas implementaciones, teniendo todas un número de ciclos por cuaternión ínfimos en comparación a la implementación por hilos de OMP. Esto se debe a que el coste de la creación de hilos es muy alto, y el número de operaciones que se requieren realizar (viene dado por *q*) es muy bajo y no compensa ese sobrecoste, resultando poco eficiente. Además de que a medida que aumentan el número de hilos que actúan, también aumentan los ciclos de forma exponencial, ya que hay un costo mayor al tener que crear más hilos.

En la gráfica casi no se pueden apreciar las diferencias entre el resto de casos, debido a los altos valores producidos por OMP que la desestabilizan, sin embargo, se añadió notación numérica para poder realizar una valoración en buenas condiciones.

Observando los valores, en cuanto a las implementaciones secuenciales, se puede distinguir como en el caso del base con optimizaciones del compilador se mejora el doble con respecto a la compilación sin optimizaciones, lo que resulta lógico. Sin embargo, ambas parecen ser menos eficientes que el secuencial optimizado (no por el compilador), debido a que las operaciones se realizan de forma más eficiente: las operaciones entre un par de cuaterniones ocurren todas en una misma iteración; sólo se realiza un bucle, aunque, cabe destacar, que no hay tanta diferencia respecto al base con optimizaciones del compilador.

En las implementaciones con extensiones de AVX, para ambos métodos, la vectorización de la multiplicación y de las iteraciones del bucle, se puede observar una cantidad de ciclos semejante.

Para este caso, se puede afirmar que las mejores implementaciones resultan ser la secuencial base optimizada (por el compilador) y la secuencial optimizada. Por otro lado, las peores son las de OMP, siendo peor cuanto más hilos se utilicen.

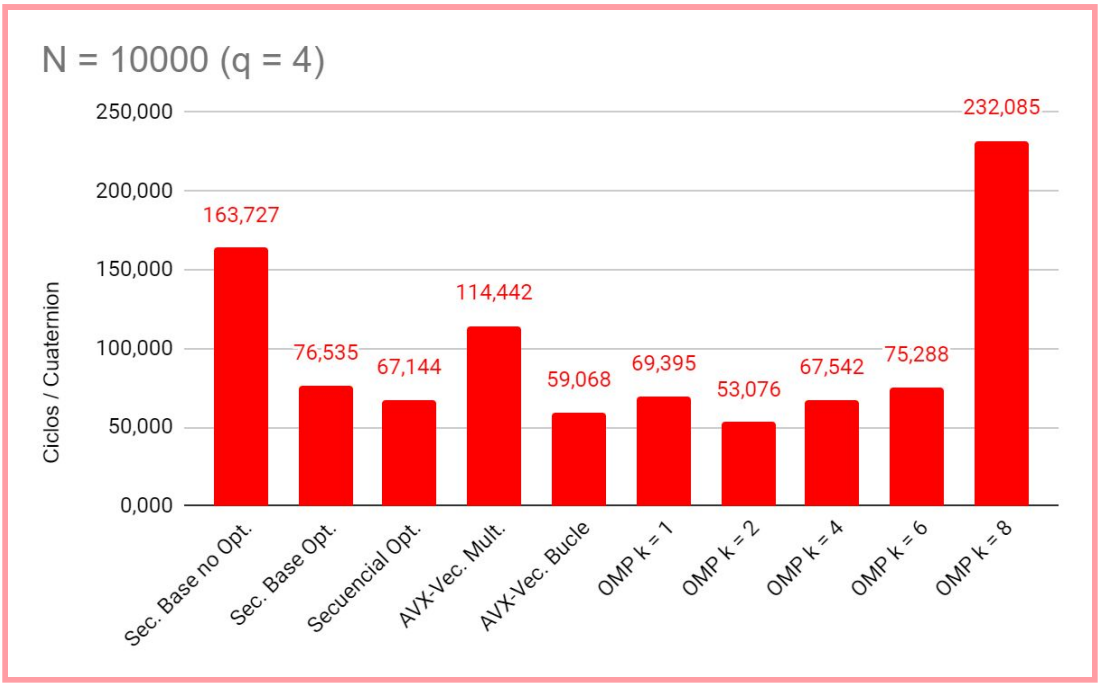

Figura 2. ciclos/cuaternión para cada implementación con *q*=4.

Para el caso de *q=4*, el coste de la creación de hilos ya parece ser más asumible en términos de entre *2* y *6* hilos, ya que el número de operaciones que se tienen que realizar compensan el gasto que produce la creación de los hilos. Sin embargo, para 8 hilos, sigue dando un número de ciclos por cuaternión muy elevado, resultando ineficiente. Además, también cabe recalcar que justamente para *2* hilos, el código resulta ser eficiente, no solo en comparación con el resto de pruebas de OMP, sino también con respecto al resto de implementaciones.

En cuanto a las implementaciones secuenciales, siguen la misma estructura que en la anterior gráfica, por lo que no hay mucho más que decir al respecto. Sin embargo, ahora resulta ser mucho más apreciable la diferencia que hay con respecto a la implementación secuencial base no optimizada por el compilador, la cual parece ser bastante ineficiente, en comparación con el resto.

Sin embargo, un cambio que resulta más apreciable y necesario comentar, es que, en este caso, en las implementaciones con extensiones de AVX, resulta ser mayor la diferencia entre la vectorización de la multiplicación y por iteraciones del bucle. Para el primer caso, comentar que el número de ciclos se ve condicionado por las funciones shuffle utilizados, los cuales añaden una carga computacional adicional al tener que reordenar los elementos de los vectores. Sin embargo, en el segundo caso, son las operaciones del tipo *fmadd* las que afectan al rendimiento. En resumen, decir que en la vectorización por iteraciones del bucle resulta ser menor que en el otro caso, por lo que es más eficiente, ya que comparándolo con el resto de implementaciones sale muy favorable.

Para este caso, se puede afirmar que entre las mejores implementaciones, se suman la vectorización por iteraciones del bucle y OMP con *2* hilos. Por otra parte, las peores en este caso son la secuencial base no optimizada y OMP con *8* hilos.

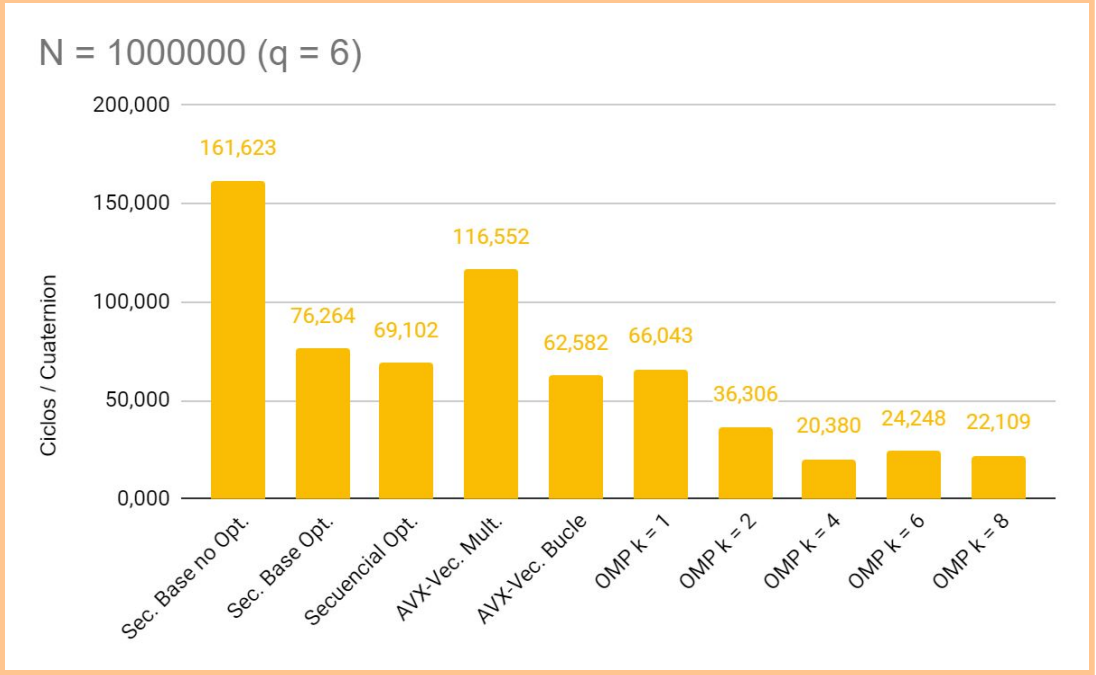

Figura 3. ciclos/cuaternión para cada implementación con *q*=6.

Para el caso de *q=6*, la mayoría de implementaciones mantienen un número de ciclos bastante similar con respecto a las mediciones representadas en la anterior gráfica, sin embargo, cabe recalcar, que ahora resulta mucho más rentable el uso de hilos con OMP, porque como ya se comentó anteriormente, el elevado coste que tiene la creación de los hilos tiene que verse compensado, en este caso, por una alta cantidad de operaciones realizadas, es decir, que existan un elevado número de cuaterniones sobre el que realizar la computación, que es lo que ocurre. Por lo tanto, ahora se puede recomendar el uso de unas determinadas implementaciones con respecto a otras, como es el caso de *4* hilos.

Para este caso, se puede afirmar que entre las mejores implementaciones, se encuentran aquellas que empiezan a repartirse entre ellos las tareas de la computación, es decir, las de OMP. Sin embargo, al igual que en el caso anterior, la peor seguirá siendo la secuencial base no optimizada.

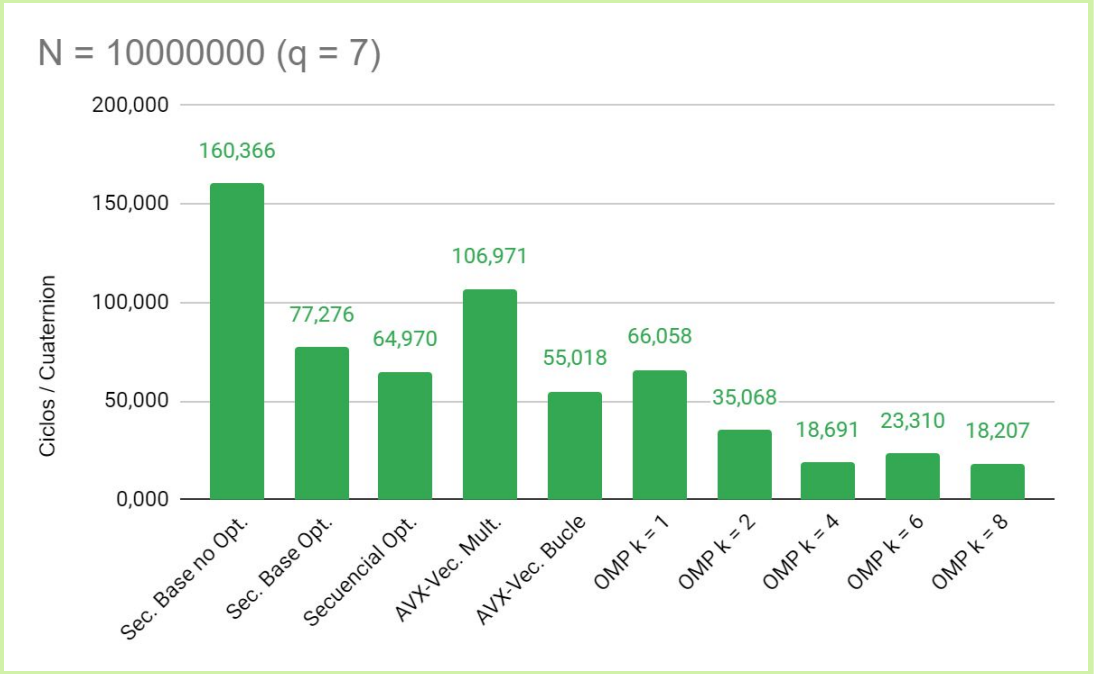

Figura 4. ciclos/cuaternión para cada implementación con *q*=7.

Por último, se analiza la situación para el caso de *q*=7, la cual se trata de una situación similar a la anterior, ya que el número de ciclos se han mantenido aproximadamente a la misma altura. Sin embargo, se potencia todavía más la efectividad de los hilos (debido al aumento de *q*), siendo más notable la diferencia de la repartición de tareas entre muchos hilos. Como era de esperar, en el caso de 8 hilos ya se reparten muy bien las operaciones, siendo el tiempo mínimo de este caso y, por tanto, el de mayor rendimiento.

Para este último caso, al igual que en el caso anterior de q=4, se puede afirmar que OMP sigue con su tendencia positiva en cuanto al número de ciclos/cuaternión existente. Lo mismo ocurre en el caso de las peores que, al igual que en los dos casos anteriores, la peor seguirá siendo la secuencial base no optimizada.

10. Conclusiones finales.

Como se pudo observar en cada una de las representaciones gráficas, la situación para cada implementación en cuanto al número de ciclos por cuaternión resulta ser muy diferente. La etapa en la que se nota una gran mejora a medida que se aumentan los tamaños de los vectores de cuaterniones existentes es en el uso de hilos de OpenMp, donde sus ciclos se ven reducidos enormemente, ya que el elevado número de operaciones compensa el alto coste de la creación de hilos. En el resto de casos, la situación se mantiene más o menos constante para cada uno de los diferentes valores de *q*.

Si se quisiera utilizar alguna de estas implementaciones para un caso de *N* bajo, sin duda alguna se elegiría la versión del secuencial optimizado. Su número de ciclos por cuaternión es bajo, más que las otras dos implementaciones secuenciales, y además no requiere de mucha complejidad para ser implementada. El resto de casos no tienen un valor de ciclos muy alto en comparación, así que tampoco sería tan disparatado usar cualquiera de esas implementaciones. Lo que está claro es que no se debe usar OpenMP en estos casos debido al alto coste de la creación de hilos, el cual no se ve compensado.

No obstante, para un valor alto de *N*, la mejora de OpenMP es muy útil, siendo con mucha diferencia la mejor implementación. Otro de los casos más destacables sería el del uso de extensiones AVX para vectorizar por iteraciones del bucle, donde no se acerca a OpenMP, pero sí que resulta una opción más favorable en comparación con el resto. Sin embargo, la complejidad que requiere su implementación podría darla por descartada. En este caso, la mejor implementación para valores altos de *N* sería, sin duda alguna, OpenMp.

11. Anexo.

Ejemplo de makefile utilizado

```
CC = gcc -Wall

MAIN = ejecutable

SOURCES = file_name.c rutinas_clock.c

LIBS = rutinas_clock.h

$(MAIN): $(SOURCES) $(LIBS)
	$(CC) -o $(MAIN) $(SOURCES) -lm -O0
```

Script de ayuda para la medición de tiempos

```
#!/bin/bash

make

sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
sudo perf stat -d ./ejecutable res.dat
```

12. Bibliografía.

- [1] Introducción a OpenMP de Elisardo Antelo Suárez, última visita 10 de mayo de 2020.
- [2] Tutorial 1 sobre AVX - Design of Parallel and High-Performance Computing de : Tal Ben-Nun & Markus Püsche, última visita 10 de mayo de 2020.
- [3] Tutorial 2 sobre AVX.
- [4] Cuaternión - Wikipedia, https://es.wikipedia.org/wiki/Cuaternión, última visita 10 de mayo de 2020.
- [5] Why use a _mm_malloc?, https://stackoverrun.com/es/q/8984458, última visita 10 de mayo de 2020.