Optimización de computación con matrices mediante diferentes técnicas

Pablo Souto Soneira, Nicolás Vilela Pérez

Laboratorio de Arquitectura de Computadores
Grupo 4
{pablo.souto.soneira, nicolas.vilela}@rai.usc.es

7 de mayo de 2021

Resumen

Se realiza una aproximación superficial a la optimización de computación con matrices mediante un ejemplo concreto. Las técnicas empleadas para la optimización son tres: desenrollamiento de bucles, extensiones vectoriales y la API OpenMP. Las diferentes gráficas y speedups resultantes nos permiten valorar el rendimiento de cada técnica, mostrando que la más eficiente para este ejemplo concreto es la que usa la API OpenMP.

Palabras clave: computación con matrices, unrolling, extensiones vectoriales, OpenMP, optimización, speedups

I. Introducción

La principal medida de eficiencia de un programa es el tiempo necesario para que se ejecute, por ello es de gran importancia aplicar modificaciones al código que permitan reducir su tiempo de CPU. Algunas de estas modificaciones, y las que serán tratadas en este informe, son la utilización de varios hilos, la utilización de extensiones vectoriales SIMD y estrategias para reducir los fallos caché.

La máquina usada para este estudio tiene las siguientes características:

- **Procesador:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- **Nº de núcleos:** 6
- **Nº de hilos:** 12
- **Tamaño RAM:** 16 GB
- **Tamaño caché L1d por CPU:** 32 KiB
- **Tamaño caché L2 por CPU:** 256 KiB
- **Tamaño caché L3:** 12 MiB
- **Tamaño línea caché:** 64 B

Lo buscado en este informe es mostrar la diferencia de latencia entre diferentes versiones de un mismo código que realiza un cálculo sobre los elementos de una matriz. Estas versiones se diferencian en que a cada una de ellas se le aplica una optimización distinta.

II. Código inicial

Para llevar a cabo el estudio ha sido necesaria la creación de un programa en C que utiliza una matriz como medio sobre el que realizar la medición. Consiste en una multiplicación entre dos matrices, en la que también interviene un vector. El resultado final es la suma de la división de todos los valores de la matriz resultante entre 2, accediendo a ésta con índices ordenados de forma aleatoria.

Se inserta aquí la computación comentada:

```
for (l = 0; l < NUMVECES; l++) {
    f = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            d[i][j] = 0;
        }
    }

    start_counter();

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < 8; k++) {
                d[i][j] += 2 * a[i][k] * (b[k][j] - c[k]);
            }
        }
    }

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

    clocks[l] = get_counter();

    printf("[Rep. %d] Ciclos: %1.10lf\tf = %lf\n", l, clocks[l], f);
}
```

III. Realización de las optimizaciones

Se procede a la aplicación de diferentes optimizaciones que se centrarán en el bucle que realiza casi toda la carga computacional.

A. Mejoras de caché

Partimos del código inicial y lo primero es analizar que optimizaciones de las posibles son las que podemos aplicar al bucle `for`.
En un primer momento probamos una de las que se nos ofrecía: el *blocking*. Esta técnica no surtió el efecto deseado y recurrimos a otra técnica conocida como *unrolling*, la cual consiste en desenrrollar el ciclo del bucle para incluir varias iteraciones del bucle en una, reduciendo o eliminando las instrucciones que controlan el bucle como puede ser la comprobación de finalización del bucle o reduciendo el número de paradas por iteración.

La aplicación de esta técnica da lugar a una versión secuencial optimizada del código inicial.

El bucle de la computación quedaría de la siguiente forma:

```
for (l = 0; l < NUMVECES; l++) {
```

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

        start_counter();

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

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

        clocks[l] = get_counter();

        printf("[Rep. %d] Ciclos: %1.10lf\tf = %lf\n", l, clocks[l], f);
}
```

B. Extensiones vectoriales

Continuamos la experimentación aplicando instrucciones vectoriales de tipo SSE, las cuales permiten relaizar operaciones sobre varios datos de forma simultánea, en concreto para este apartado lo hacemos sobre 2 valores. Es importante tener en cuenta que para utiliar estas instrucciones se necesita alinear la reserva de memoria con el inicio de la línea caché y que fue necesaria la trasposición de la matriz **b** para multiplicar de forma correcta las matrices.
Este tipo de operaciones requieren que los datos que se van a emplear sean previamente cargados con la instrucción `_mm_set_pd`, la cual coge el valor que esté en la dirección que se pasa como argumento y el siguiente a este. Los almacenamientos de los resultados también requieren de una operación especial para ser llevados a memoria (`_mm_store_pd`).

Tras aplicar las extensiones vectoriales el bucle tiene el siguiente aspecto:

```
for (l = 0; l < NUMVECES; l++) {
    f = 0;

    start_counter();

    __m128d rA, rB, rC, rD, vec2, valor;
    vec2 = _mm_set_pd(2.0, 2.0);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j += 2) {
            rD = _mm_set_pd(0.0, 0.0);
            _mm_store_pd(d[i] + j, rD);
            for (k = 0; k < 8; k += 2) {
                rA = _mm_load_pd(a[i] + k);
                rB = _mm_load_pd(bt[j] + k);
                rC = _mm_load_pd(c + k);
```

```
                valor = _mm_sub_pd(rB,  rC);
                valor = _mm_mul_pd(rA, valor);
                valor = _mm_mul_pd(valor, vec2);

                rD = _mm_add_pd(rD, valor);


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

                valor = _mm_sub_pd(rB, rC);
                valor = _mm_mul_pd(rA, valor);
                valor = _mm_mul_pd(valor, vec2);

                rD = _mm_add_pd(rD, valor);


                //d[i][j] += 2 * a[i][k] * (b[k][j] - c[k]);
            }
        _mm_store_pd(d[i] + j, rD);
        }
    }

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

    clocks[l] = get_counter();

    printf("[Rep. %d] Ciclos: %1.10lf\tf = %lf\n", l, clocks[l], f);
}
```

C. OpenMP

La mejor forma de conseguir reducir los ciclos de un código es mediante el paralelismo. La API OpenMP facilita la implementación de este paralelismo, ofreciendo una serie de directivas de compilador a través de las cuales se puede paralelizar una zona de código determinada para que se ejecuten varios hilos a la vez, reduciendo de forma significativa los ciclos empleados.

El bucle queda de la siguiente forma tras la implementación con OpenMP:

```
for (l = 0; l < NUMVECES; l++) {
    f = 0;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            d[i][j] = 0;
        }
    }

    start_counter();

    #pragma omp parallel private(i,j) num_threads(T)
    {
        #pragma omp for
            for (i = 0; i < N; i++) {
                    for (j = 0; j < N; j++) {
                        d[i][j] += 2 * a[i][0] * (b[0][j] - c[0]);
                        d[i][j] += 2 * a[i][1] * (b[1][j] - c[1]);
                        d[i][j] += 2 * a[i][2] * (b[2][j] - c[2]);
                        d[i][j] += 2 * a[i][3] * (b[3][j] - c[3]);
                        d[i][j] += 2 * a[i][4] * (b[4][j] - c[4]);
                        d[i][j] += 2 * a[i][5] * (b[5][j] - c[5]);
```

```
                        d[i][j] += 2 * a[i][6] * (b[6][j] - c[6]);
                        d[i][j] += 2 * a[i][7] * (b[7][j] - c[7]);
                    }
            }
    }

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

    clocks[l] = get_counter();

    printf("[Rep. %d] Ciclos: %1.10lf\tf = %lf\n", l, clocks[l], f);
}
```

IV. Resultados

Tras la finalización de la experimentación se recurre a las siguientes gráficas para explicar los resultados obtenidos:

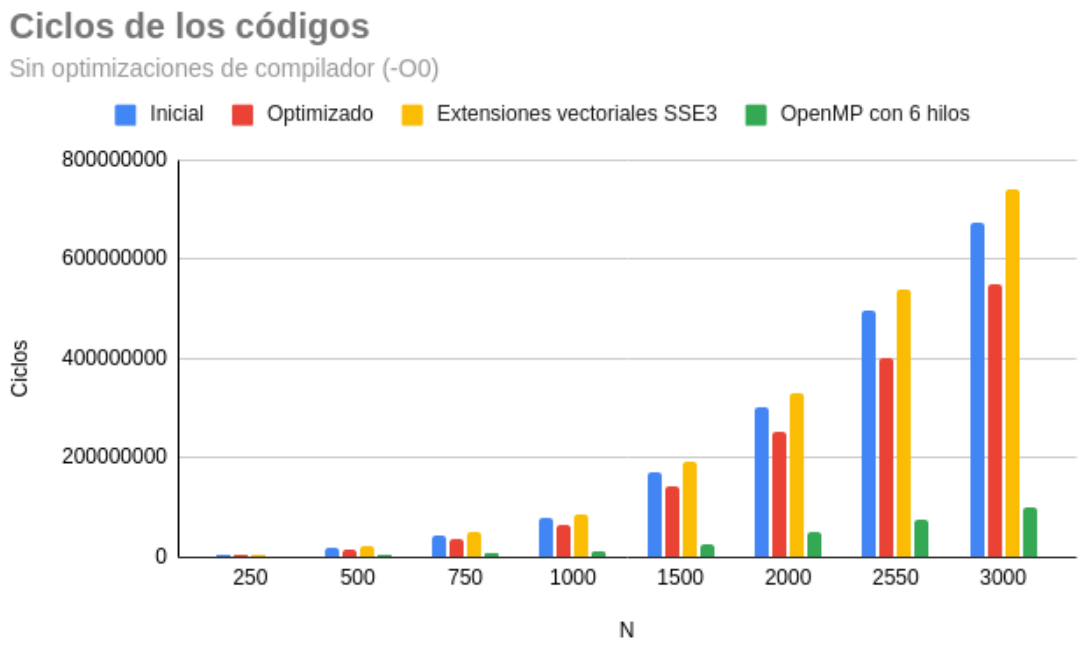

Figura 1: *Ciclos de las diferentes versiones*

Como se puede observar las diferencias entre las 3 estrategias se acentúan a medida que aumentamos **N**. Sin embargo, para todos los valores de **N** el orden en cuanto a número de ciclos de los tres métodos se mantiene constante, siendo el orden de más a menos ciclos (y por tanto de menos a más rápido): extensiones vectoriales, versión inicial, secuencial optimizado y OpenMP.

En un primer momento puede resultar extraño que las extensiones vectoriales ralenticen la ejecución pero esto se justifica de la siguiente forma: las extensiones vectoriales son buenas para la multiplicación de grandes vectores ya que el tiempo que requiere cargar los valores se compensa con el tiempo que ahorras en cálculo, sin embargo, al usar como tipo de

dato `__m128d` solamente podemos cargar dos valores y por tanto el tiempo que ahorramos en cálculo se ve eclipsado, e incluso superado, por el tiempo que requiere la carga de los valores.

El apartado de OpenMP se realiza con diferentes números de hilos ya que esta API está diseñada para aprovechar el paralelismo que ofrecen los hilos mediante la distribución de la carga computacional.

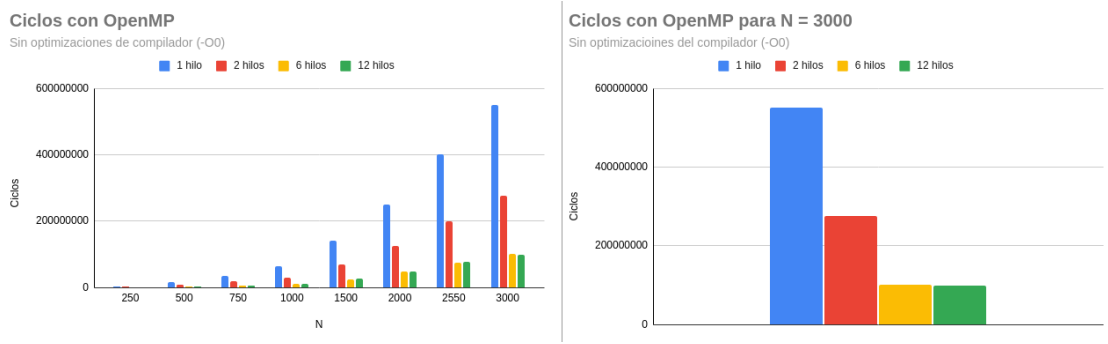

Figura 2: *Ciclos con OpenMP*

Como vemos, el aumento del número de hilos disminuye el número de ciclos ya que estos se reparten entre los diferentes hilos, sin embargo esta mejora no se aplica al caso de los 12 hilos. Una posible explicación podría ser que aunque el hardware empleado soporte el uso de 12 hilos (12 CPUs lógicas), el sistema cuenta únicamente con 6 CPUs físicas, por lo que cada una de ellas ejecuta dos hilos y por tanto divide su capacidad entre los dos, limitando así la mejora obtenida.

Puede resultar de interés comparar diferentes **speedups** de los códigos.
La versión secuencial optimizada presenta un speedup de 1.22 sobre el código inicial. Esto demuestra que el unrolling ha tenido efecto.
La versión con extensiones vectoriales SSE3 presenta un speedup inferior a 1 respecto a la versión secuencial optimizada (0.74), demostrando así que no son eficientes para este caso por lo comentado anteriormente.
La versión con OpenMP con 6 hilos sin optimizaciones de compilador presenta un speedup de 5.45 relativo a la versión secuencial optimizada, mientras que con optimizaciones de compilador -O2 presenta un speedup de 26.71 sobre la versión secuencial optimizada, también. Queda así más que demostrado que los hilos reducen mucho el tiempo de computación debido al paralelismo que ofrecen.

Por último se comparan el código inicial con optimizaciones de compilador -O3 frente a los anteriores (para saber que optimizaciones tiene esta opción, o cualquier otra, se puede ver en [1].)
Sobre la versión secuencial optimizada presenta un speedup de 14.41; sobre la versión con extensiones vectoriales SSE3 presenta un 19.36; sobre la versión con OpenMP con 6 hilos sin optimizaciones, un 2.64; y sobre la versión con OpenMP con 6 hilos con optimizaciones de compilador -O2, un 0.54 (inferior que 1).

De estos últimos speedups se puede concluir que la mejor versión de todas las testadas es la versión con OpenMP con optimizaciones de compilador -O2, bien sea con 6 o con 12 hilos. Es más recomendable usar 6 hilos, ya que con 12 no hay una mejora significativa y acapara muchos más recursos de CPU.

V. Conclusiones

La optimización de códigos es un campo de gran importancia y muy investigado ya que, aunque en nuestro caso no suponga mucha diferencia de tiempo (unos pocos milisegundos), en programas de gran carga computacional reducir el número de ciclos por iteración, aunque sea mínimamente, puede marcar una gran diferencia en el tiempo de ejecución cuando tratamos con millones de iteraciones.

Atendiendo a las gráficas presentadas en el apartado anterior y en especial a los **speedups** obtenidos podemos concluir que contamos con muchas y muy diversas herramientas para la optimización de código pero todas tienen en común que se requiere de las condiciones adecuadas para aplicarlas ya que, por ejemplo como ocurre con las extensiones vectoriales, de no ser así, su uso puede incluso llegar a perjudicar el rendimiento.

La mejor versión de las testadas es la que implementa la API OpenMP usando 6 hilos y optimizaciones de compilador -O2.

Por último reslatar que este estudio se queda muy pequeño dentro de la optimización de código y podría ser ampliado y complementado de diversas formas, ya sea profundizando más en las herramientas aquí tratadas o escogiendo otras.

Referencias

[1] Opciones de optimización de GCC
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [online] última visita abril de 2021.