Compiladores e Interpretes

Práctica 2

Optimización de Búsqueda de Subexpresiones Locales Comunes

19 de diciembre de 2021

Contenido

Introducción

En este trabajo vamos a realizar una medición sobre la mejora de rendimiento de un código aplicandole la optimización de búsqueda de subexpresiones locales comunes.

El código base que será usado para las mediciones se recoge en la siguiente figura. Este depende de los valores que tomen N e ITER.

```
int i, j;

float a, b, c, d, k, x, y, z, t;
float A[N];
for (j = 0; j < ITER; j++)
    for (i = 0; i < N; i++) {
        a = x * y * z * t * A[i];
        b = x * z * t;
        c = y * t * A[i];
        d = k * A[i] - x * z;
        A[i] = (z * x * A[i]) + (y * t) + A[i];
    }

float tmp1, tmp2, tmp3, tmp4;
for (j = 0; j < ITER; j++)
    for (i = 0; i < N; i++) {
        tmp1 = x * z;
        tmp2 = A[i];
        tmp3 = y * t;
        tmp4 = tmp1 * tmp2;
        a = tmp3 * tmp4;
        b = tmp1 * t;
        c = tmp3 * tmp2;
        d = k * tmp2 - tmp1;
        A[i] = tmp4 + tmp3 + tmp2;
    }
```

Código 1 - Código de la implementación no optimizada y optimizada manualmente

Las características de la máquina utilizada para realizar las pruebas de rendimiento, dispuestas en los siguiente apartados, están dispuestas en el apartado Anexo - Máquina de pruebas.

Estudio de la optimización

Código fuente

Una primera visión de las diferencias entre los códigos se nos ofrece comparando los dos códigos fuente en C.

Compararemos los dos códigos según las instrucciones que los componen y daremos una primera conclusión sobre la futura diferencia de eficiencia entre códigos.

En la siguiente figura se representan las diferencias de los códigos, separados por líneas y grupos de instrucciones.

CÓDIGO SIN OPTIMIZAR	CÓDIGO OPTIMIZADO
1	
`for (j = 0; j < ITER; j++)` `  for (i = 0; i < N; i++) {`	`for (j = 0; j < ITER; j++)` `  for (i = 0; i < N; i++) {`
2	

	`tmp1 = x * z;`
	`tmp2 = A[i];`
	`tmp3 = y * t;`
	`tmp4 = tmp1 * tmp2;`

3	

`a = x * y * z * t * A[i];`
`b = x * z * t;`
`c = y * t * A[i];`
`d = k * A[i] - x * z;`
	`a = tmp3 * tmp4;`
	`b = tmp1 * t;`
	`c = tmp3 * tmp2;`
	`d = k * tmp2 - tmp1;`

4	
`A[i] = (z * x * A[i]) + (y * t) + A[i];` `  }`	`A[i] = tmp4 + tmp3 + tmp2;` `}`

Código 2 - Comparación de código por líneas

Las diferencias observables están en los grupos de instrucciones 2 y 3 del código optimizado respecto del no optimizado.

En el grupo 2 del código optimizado se realizan aquellas operaciones, del grupo 3 del código no optimizado, que se repiten entre el cálculo de las variables `a`, `b`, `c` y `d`, intentando disminuir el tiempo de computación asociado.

En la agrupación 3 es donde se observan directamente las diferencias. El cálculo de las variables en el código no optimizado repite operaciones múltiples veces, siendo el código optimizado el que disminuye el número de cálculos intermedios usando resultados calculados anteriormente y guardados en variables temporales.

Por las aparentes mejoras que conlleva esta modificación del código, se espera que disminuya el tiempo computacional del programa, al evitar realizar más operaciones de las necesarias.

Código ensamblador

A raíz del código anterior no podemos observar como se reparten los registros ni las instrucciones de carga y guardado de datos. Para obtener esta información es necesario revisar el código ensamblador.

A continuación, se muestran las líneas de código del programa en C con sus correspondientes instrucciones en lenguaje ensamblador del MIPS64 y una explicación de la instrucción.

CODIGO SIN OPTIMIZAR			
`a = x * y * z * t * A[i];`	`lwc1`	`$f1,16($fp)`	**Cargar x**
	`lwc1`	`$f0,20($fp)`	**Cargar y**
	`mul.s`	`$f1,$f1,$f0`	aux = x * y
	`lwc1`	`$f0,24($fp)`	**Cargar z**
	`mul.s`	`$f1,$f1,$f0`	aux = aux * z
	`lwc1`	`$f0,28($fp)`	**Cargar t**
	`mul.s`	`$f1,$f1,$f0`	aux = aux * t
	`lw`	`$2,0($fp)`	**Cargar i**
	`dsll`	`$2,$2,2`	Shift left lógico i
	`ld`	`$3,8($fp)`	**Cargar &A (2 instr.)**
	`daddu`	`$2,$3,$2`	pos = &A + i
	`lwc1`	`$f0,0($2)`	**Cargar pos (A[i])**
	`mul.s`	`$f0,$f1,$f0`	a = aux * A[i]
	`swc1`	`$f0,32($fp)`	**Guardar a en el frame**
`b = x * z * t;`	`lwc1`	`$f1,16($fp)`	**Cargar x**
	`lwc1`	`$f0,24($fp)`	**Cargar z**
	`mul.s`	`$f1,$f1,$f0`	aux = x * z
	`lwc1`	`$f0,28($fp)`	**Cargar t**
	`mul.s`	`$f0,$f1,$f0`	b = aux * t
	`swc1`	`$f0,36($fp)`	**Guardar b en el frame**
`c = y * t * A[i];`	`lwc1`	`$f1,20($fp)`	**Cargar y**
	`lwc1`	`$f0,28($fp)`	**Cargar t**
	`mul.s`	`$f1,$f1,$f0`	aux = y * t
	`lw`	`$2,0($fp)`	**Cargar i**
	`dsll`	`$2,$2,2`	Shift left lógico i
	`ld`	`$3,8($fp)`	**Cargar &A (2 instr.)**
	`daddu`	`$2,$3,$2`	pos = &A + i
	`lwc1`	`$f0,0($2)`	**Cargar pos (A[i])**
	`mul.s`	`$f0,$f1,$f0`	c = aux * A[i]
	`swc1`	`$f0,40($fp)`	**Guardar c en el frame**
`d = k * A[i] - x * z;`	`lw`	`$2,0($fp)`	**Cargar i**

	dsll	$2,$2,2	Shift left lógico i
	ld	$3,8($fp)	**Cargar &A (2 instr.)**
	daddu	$2,$3,$2	pos = &A + i
	lwc1	$f1,0($2)	**Cargar pos (A[i])**
	lwc1	$f0,44($fp)	**Cargar k**
	mul.s	$f1,$f1,$f0	aux1 = A[i] * k
	lwc1	$f2,16($fp)	**Cargar x**
	lwc1	$f0,24($fp)	**Cargar z**
	mul.s	$f0,$f2,$f0	aux2 = x * z
	sub.s	$f0,$f1,$f0	d = aux1 - aux2
	swc1	$f0,48($fp)	**Guardar d en el frame**
A[i] = (z * x * A[i]) + (y * t) + A[i];	lw	$2,0($fp)	**Cargar i**
	dsll	$2,$2,2	Shift left lógico i
	ld	$3,8($fp)	**Cargar &A (2 instr.)**
	daddu	$2,$3,$2	pos1 = &A + i
	lwc1	$f1,24($fp)	**Cargar z**
	lwc1	$f0,16($fp)	**Cargar x**
	mul.s	$f1,$f1,$f0	aux1 = z * x
	lw	$3,0($fp)	**Cargar i**
	dsll	$3,$3,2	Shift left lógico i
	ld	$4,8($fp)	**Cargar &A (2 instr.)**
	daddu	$3,$4,$3	pos2 = &A + i
	lwc1	$f0,0($3)	**Cargar pos2 (A[i])**
	mul.s	$f1,$f1,$f0	aux1 = aux1 * A[i]
	lwc1	$f2,20($fp)	**Cargar y**
	lwc1	$f0,28($fp)	**Cargar t**
	mul.s	$f0,$f2,$f0	aux2 = y * t
	add.s	$f1,$f1,$f0	aux1 = aux1 + aux2
	lw	$3,0($fp)	**Cargar i**
	dsll	$3,$3,2	Shift left lógico i
	ld	$4,8($fp)	**Cargar &A (2 instr.)**
	daddu	$3,$4,$3	pos3 = &A + i
	lwc1	$f0,0($3)	**Cargar pos3 (A[i])**
	add.s	$f0,$f1,$f0	A[i] = aux1 + A[i]
	swc1	$f0,0($2)	**Guardar A[i] en pos1**
ACCESOS A MEMORIA TOTALES: 44			

Ensamblador 1 - Código no optimizado comentado por líneas

CODIGO OPTIMIZADO			
tmp1 = x * z;	lwc1	$f1,16($fp)	**Cargar x**
	lwc1	$f0,20($fp)	**Cargar z**
	mul.s	$f0,$f1,$f0	tmp1 = x * z
	swc1	$f0,24($fp)	**Guardar tmp1 en el frame**
tmp2 = A[i];	lw	$2,0($fp)	**Cargar i**
	dsll	$2,$2,2	Shift left lógico i
	ld	$3,8($fp)	**Cargar &A (2 instr.)**

	daddu	$2,$3,$2	pos = &A + i
	lwc1	$f0,0($2)	**Cargar pos (A[i])**
	swc1	$f0,28($fp)	**Guardar tmp2 en el frame**
tmp3 = y * t;	lwc1	$f1,32($fp)	**Cargar y**
	lwc1	$f0,36($fp)	**Cargar t**
	mul.s	$f0,$f1,$f0	tmp3 = y * t
	swc1	$f0,40($fp)	**Guardar tmp3 en el frame**
tmp4 = tmp1 * tmp2;	lwc1	$f1,24($fp)	**Cargar tmp1**
	lwc1	$f0,28($fp)	**Cargar tmp2**
	mul.s	$f0,$f1,$f0	tmp4 = tmp1 * tmp2
	swc1	$f0,44($fp)	**Guardar tmp4 en el frame**
a = tmp3 * tmp4;	lwc1	$f1,40($fp)	**Cargar tmp3**
	lwc1	$f0,44($fp)	**Cargar tmp4**
	mul.s	$f0,$f1,$f0	a = tmp3 * tmp4
	swc1	$f0,48($fp)	**Guardar a en el frame**
b = tmp1 * t;	lwc1	$f1,24($fp)	**Cargar tmp1**
	lwc1	$f0,36($fp)	**Cargar t**
	mul.s	$f0,$f1,$f0	b = tmp1 * t
	swc1	$f0,52($fp)	**Guardar b en el frame**
c = tmp3 * tmp2;	lwc1	$f1,40($fp)	**Cargar tmp3**
	lwc1	$f0,28($fp)	**Cargar tmp2**
	mul.s	$f0,$f1,$f0	c = tmp3 * tmp2
	swc1	$f0,56($fp)	**Guardar c en el frame**
d = k * tmp2 - tmp1;	lwc1	$f1,60($fp)	**Cargar k**
	lwc1	$f0,28($fp)	**Cargar tmp2**
	mul.s	$f1,$f1,$f0	aux = k * tmp2
	lwc1	$f0,24($fp)	**Cargar tmp1**
	sub.s	$f0,$f1,$f0	d = aux - tmp1
	swc1	$f0,64($fp)	**Guardar d en el frame**
A[i] = tmp4 + tmp3 + tmp2;	lw	$2,0($fp)	**Cargar i**
	dsll	$2,$2,2	Shift left lógico i
	ld	$3,8($fp)	**Cargar &A (2 instr.)**
	daddu	$2,$3,$2	pos = &A + i
	lwc1	$f1,44($fp)	**Cargar tmp4**
	lwc1	$f0,40($fp)	**Cargar tmp3**
	add.s	$f1,$f1,$f0	aux = tmp4 + tmp3
	lwc1	$f0,28($fp)	**Cargar tmp2**
	add.s	$f0,$f1,$f0	A[i] = aux + tmp2
	swc1	$f0,0($2)	**Guardar A[i] en pos**
ACCESOS A MEMORIA TOTALES:			**34**

Ensamblador 2 - Código optimizado comentado por líneas

Después de revisar el código de los dos programas y anotar respectivamente las instrucciones que se llevan a cabo, podemos denotar una clara diferencia en el número de instrucciones de acceso a memoria.

Mientras que el programa sin optimizar accede un total de 44 veces a memoria, contando instrucciones de carga y guardado, el programa optimizado accede 34 veces (10 veces menos). Dada esta diferencia y, sumando el menor número de instrucciones de cálculo debido a su cálculo anterior, se puede pensar que será el programa optimizado el que tenga menos accesos a memoria y por tanto tenga un menor coste temporal.

Estudio sobre N e ITER

De primera mano, no sabemos como puede afectar el tamaño de N al código y, por tanto, los comportamientos de los distintos códigos pueden variar dependiendo del tamaño de esta variable. ITER, al ser una variable usada para aumentar el tiempo de computación hasta obtener resultados en el orden de segundos, también se va a ver afectado por esta variación, pues cuanto menor sea el valor asociado a N, mayor será su valor.

Primeramente, realizaremos un estudio preeliminar sobre N, de forma que su valor se variará entre tres rangos:

- **Bajo:** [2e3, 20000]
    - Salto de 2000 entre valores.
    - ITER = 400000
- **Medio:** [200000, 2000000]
    - Salto de 200000 entre valores.
    - ITER = 800
- **Alto:** [200000000, 2000000000]
    - Salto de 200000000 entre valores.
    - ITER = 1

Repetiremos cada experimento un número reducido de veces (5) para obtener un pequeño adelanto de lo que nos podemos encontrar utilizando los valores seleccionados y así realizar una buena evaluación del rendimiento.

Resultados

Los resultados obtenidos de los experimentos se han representado en varias gráficas dependiendo del rango de N utilizado.

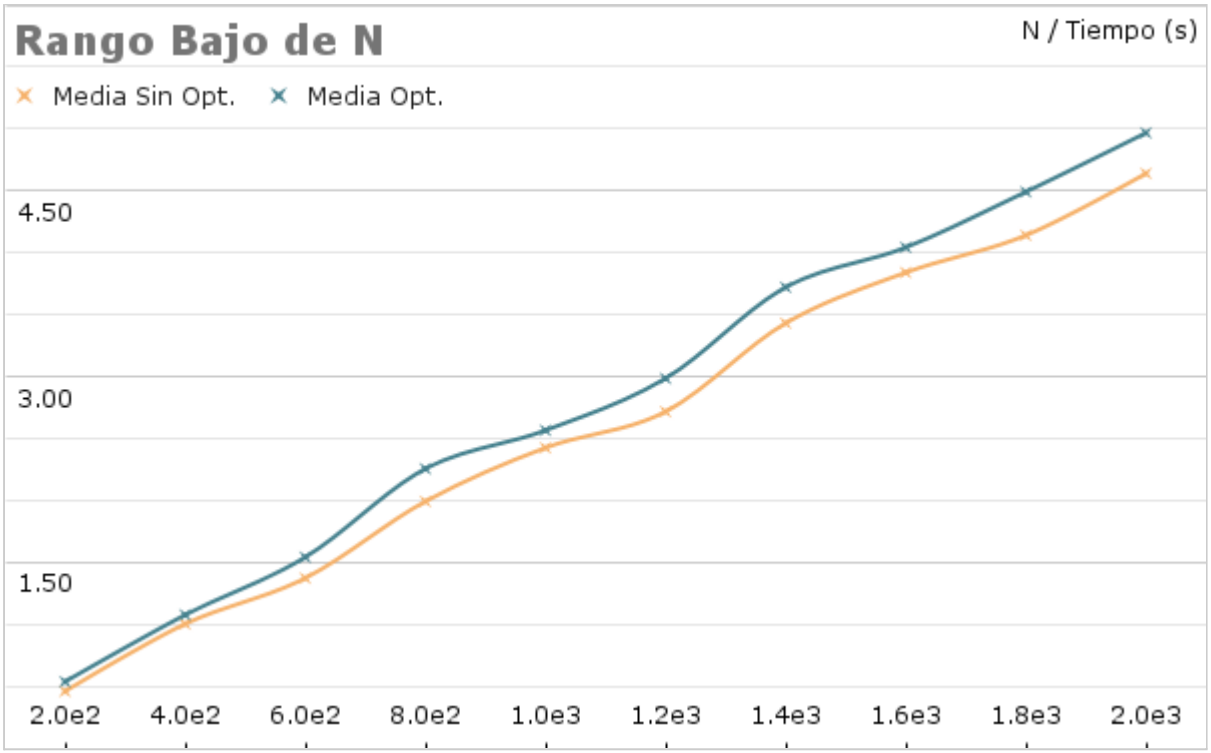

Figura 1 - Valores Bajos de N

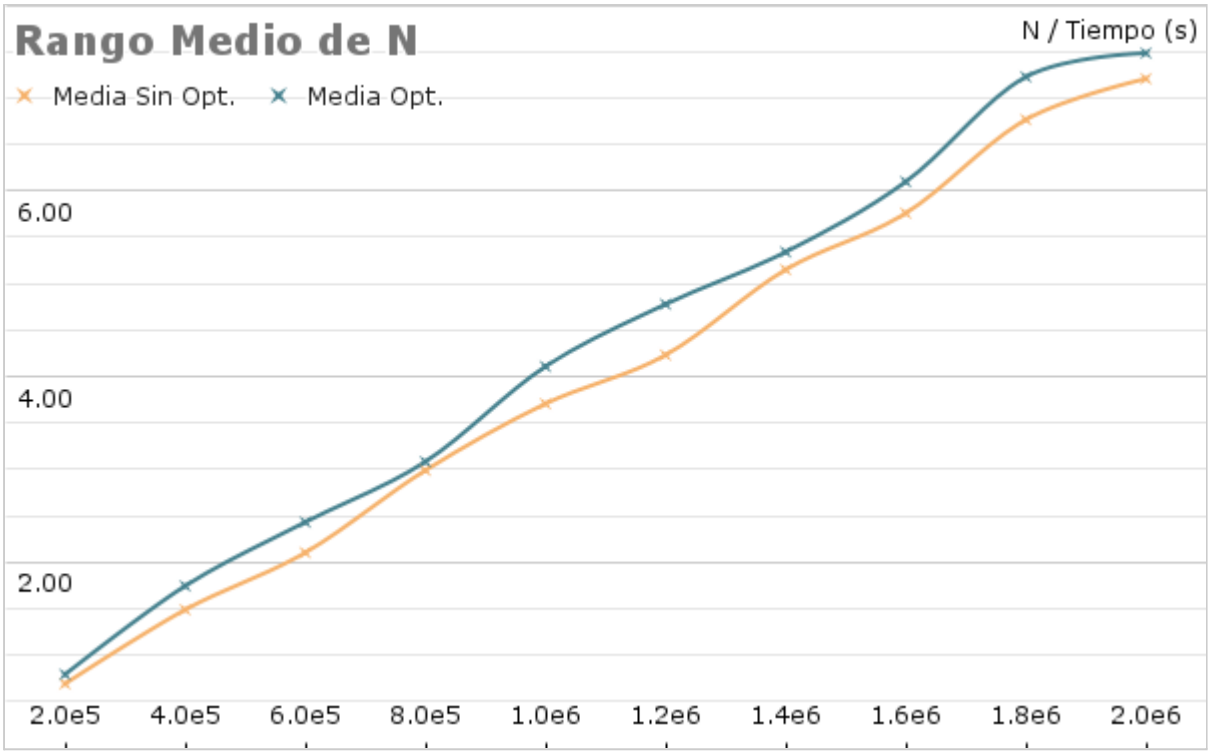

Figura 2 - Valores Medios de N

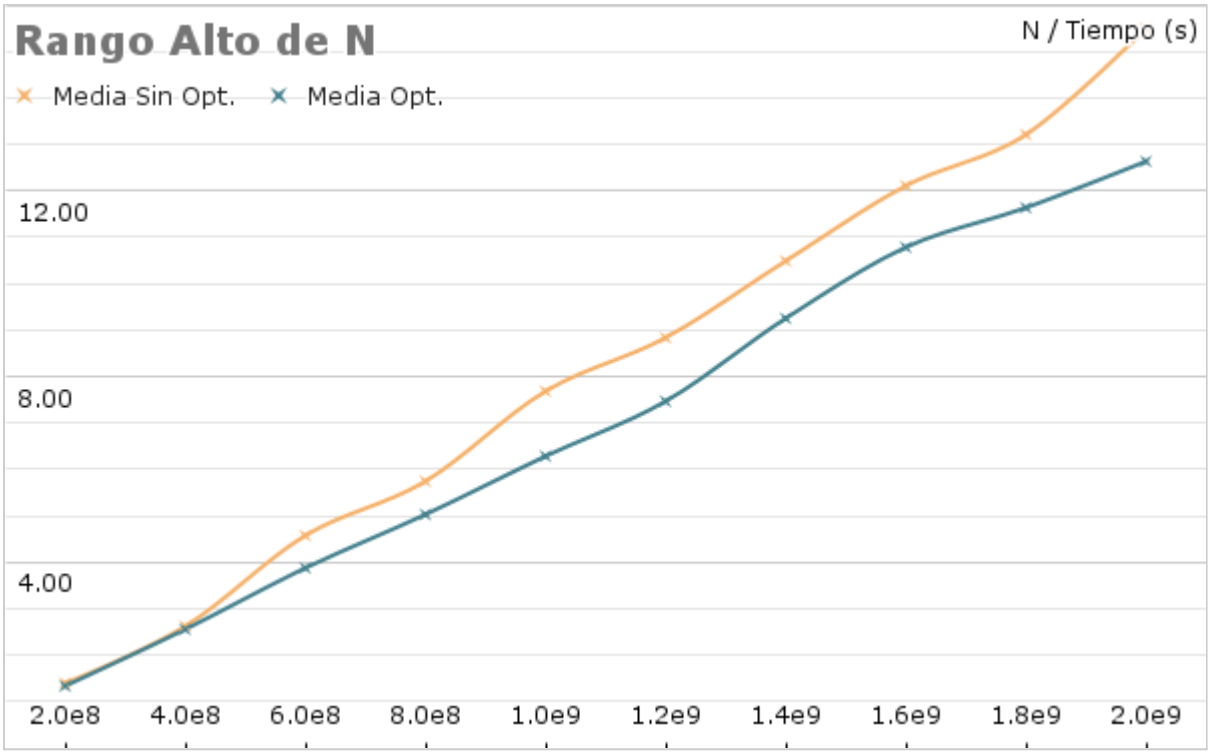

Figura 3 - Valores Altos de N

Gracias a las gráficas. observamos que para los rangos de valores del rango bajo y medio de N la implementación optimizada no presenta una mejora suficiente como para mejorar la eficiencia del codigo original. No obstante, para el rango de valores alto de N, la optimización es notable, mejorando significativamente el rendimiento del código.

Según los hechos observados, se ha decidido realizar el estudio de la mejora de rendimiento sobre un rango alto de valores de N.

Comparación de rendimiento

Tal y como fue recogido en el estudio para seleccionar los valores de N a usar, se ha seleccionado un rango de valores altos para N. El intervalo utilizado es [100000000, 2000000000] con un paso entre valores de 100000000 y un valor para ITER de 2.

Se realizaron un total de 15 pruebas para cada valor de N, tratándose los resultados obtenidos de forma posterior para descartar posibles outliers que presentes causados por las diversas acciones del sistema operativo.

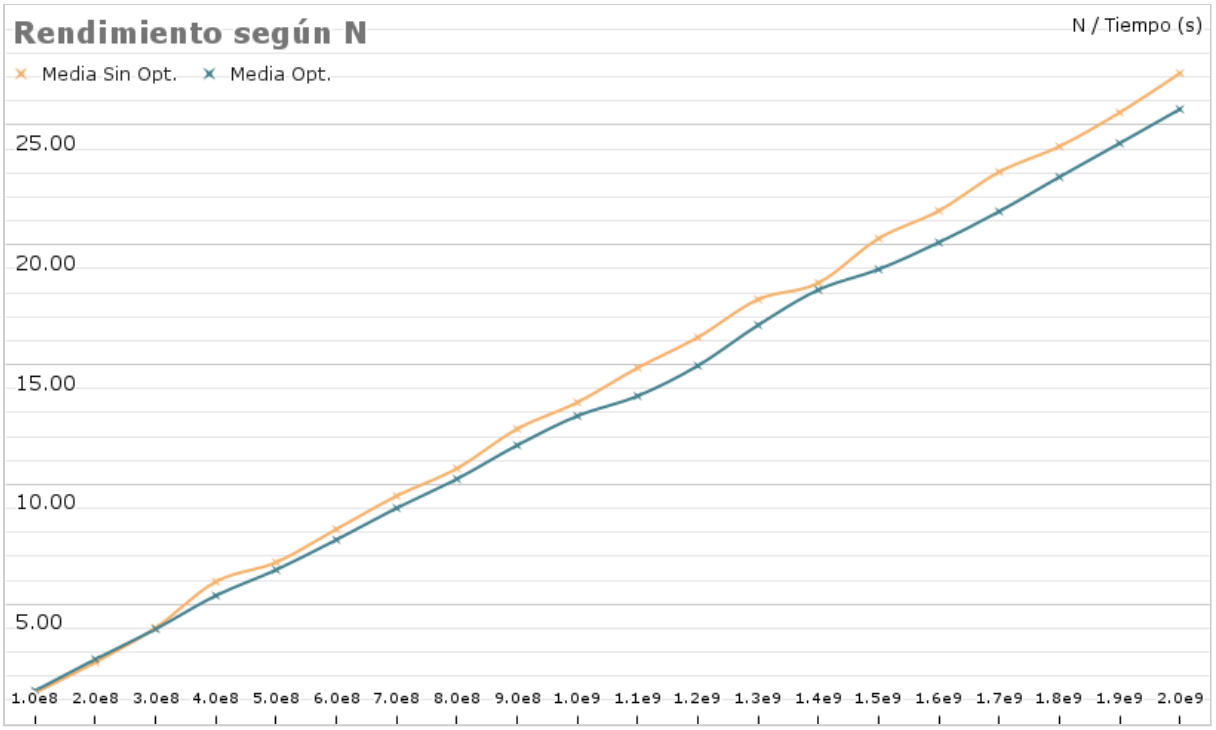

Figura 4 - Comparación de rendimiento

Se puede observar que con los primeros valores de N, el rendimiento del programa sin optimizar es ligeramente mejor que el programa no optimizado. no obstante, conforme aumenta el valor de N mejora de forma pareja el rendimiento del código optimizado respecto del código sin optimizar, llegando a tener una diferencia de algo más de un segundo con el último, y mayor, valor de N.

Conclusiones

Como se puede observar en los resultados obtenidos, con los primeros valores de N, el tiempo de ejecución del código optimizado es mayor que el del código no optimizado. Cuando N alcanza el valor de 300000000 los tiempos de ejecución se invierten, siendo el código optimizado el que se ejecuta en un menor tiempo.

Puede resultar algo confuso que no sea hasta que N alcanza valores extremadamente altos que este código no mejora el rendimiento con esta optimización, puesto que se presta bien a implementarla. No obstante, los accesos a memoria son un punto clave en la evolución de los rendimientos de los códigos.

De forma general, el código no optimizado tiene un mayor número de accesos a memoria que el código optimizado, por lo que se espera un mayor rendimiento del último. Sin embargo, al realizar esta asunción no se está teniendo en cuenta el principio de localidad. El reiterado uso de las mismas variables en instrucciones cercanas del código origen permite a la CPU aplicar el principio de localidad eficientemente, necesitando solamente unos pocos sino un

único acceso a memoria principal para obtener los datos necesarios y repartiendolos en los distintos niveles de caché disponibles.

Otro aspecto importante del código es la presencia del array `A` donde se guardan los resultados. El tamaño de este array influye en el rendimiento del código, pues es necesario el acceso a memoria para obtener los valores que contiene. El principio de localidad también influye en el rendimiento del código, pues múltiples valores del array son movidos a cache con cada acceso a memoria principal. Para valores pequeños de N es seguro que todos sino casi todos los valores del array son movidos a cache con solo un acceso a memoria principal, siendo sumamente eficiente y su tamaño no es relevante pues no ocupa una gran parte de los niveles de caché. Con los tamaños medios serán necesarios varios accesos, pero sigue sin acaparar la caché. La cosa cambia cuando los tamaños de N se mueven al rango alto, pues el array tiene tal tamaño que son necesarios múltiples accesos a memoria principal para acceder a todos los datos y la caché acaba no siendo suficiente para dar cabida a todos los datos. Esta incapacidad de la caché para guardar todos los datos puede resultar en la eliminación de datos utilizados por el ejecutable, siendo necesario otro acceso a memoria principal para obtener los datos para realizar las operaciones. El código no optimizado, al realizar operaciones con todos los datos indiscriminadamente implica que en el caso de que se elimine un dato, es casi seguro que será necesario volver a acceder a memoria para recuperarlo y necesitando eliminar otros datos disponibles, necesitando otro acceso a memoria principal en la siguiente iteración. Si ocurre con el código optimizado no implicaría tanto problema, pues al realizar las operaciones parcialmente y guardando los resultados, es casi seguro que una variable utilizada anteriormente no volverá a utilizarse en esa iteración.

En conclusión, para los valores de N bajos y medios el principio de localidad aumenta enormemente el rendimiento del código no optimizado, reduciendo el impacto del gran número de accesos a memoria de los que dispone. No obstante, para valores altos de N, el principio de localidad actúa de forma inversa debido al tamaño del array de resultados, que acaba eliminando datos necesarios por el programa en las siguientes iteraciones e implicando accesos a memoria adicionales. Este inconveniente está presente, en mayor medida, en el código no optimizado debido a el cálculo continuo y no realizar cálculos intermedios, como si se hace en el código optimizado. Es debido a estos motivos que el rendimiento del código optimizado es notablemente mejor en este rango de valores.

Bibliografía

[1] Compiler Explorer. [En línea] [Fecha de consulta 08/12/21]: https://godbolt.org/.

Anexo - Máquina de pruebas

Las pruebas se han realizado sobre un ordenador portátil, conectado a la corriente y con el máximo rendimiento, que cuenta con las siguientes características:

SO	Ubuntu 21.04
Procesador	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Arquitectura	x86_64
Cache L1d	128 KiB
Cache L1i	128 KiB
Cache L2	1 MiB
Cache L3	6 MiB
RAM	16 GiB

Anexo - Archivos de código

Los archivos de código y scripts utilizados para realizar las pruebas se pueden encontrar en la carpeta `codigo` presente en el documento comprimido. Se dispone de tres archivos:

- **rendimiento.c**
    - Archivo contenedor del código C optimizado y sin optimizar.
- **rendimiento.sh**
    - Script bash para realizar pruebas de rendimiento de los dos trozos de código.
- **estudio_n.sh**
    - Script bash para realizar un estudio sobre los valores de N.