# Práctica 2: Generación y optimización de código

*Optimización número 10: Minimizar la sobrecarga de los condicionales*

# Índice

1. Introducción....3
2. La optimización....3
3. Ejecución....4
3.1 Estudio preliminar....4
3.2 Código ensamblador....5
3.3 Resultados obtenidos....6
4. Análisis....7
5. Conclusiones....8

# 1. Introducción

En este documento se expone el análisis de una técnica de optimización, consistente en minimizar la carga que suponen los condicionales en un código. Esta técnica se prueba posteriormente utilizando el lenguaje C para obtener tiempos de ejecución y poder obtener conclusiones acerca de la optimización.

Para ello se genera un programa que ejecuta dos bucles, uno sin la optimización y otro con ella. De tal manera que se vea cómo afecta esta segunda versión a los tiempos de ejecución medidos con la función *getttimeofday*.

# 2. La optimización

La optimización se basa en eliminar los condicionales utilizados en cada iteración y que se encuentran dentro de un bucle anidado, con el objetivo de reducir el tiempo del bucle general.

El código sin optimizar es el siguiente, donde se puede ver que se guardan valores recorriendo dos matrices de tamaño 2*N filas y 2*N columnas. Y comprobando en cada iteración si se ha llegado a la posición [N, N] de la matriz, y en cuyo caso se le asignará el valor “0,0”.

```
int i, j, k;
float x[2*N][2*N], z[2*N][2*N];
for(k=0; k<ITER; k++)
   for(j=-N; j<N; j++)
      for(i=-N; i<N; i++){
         if(i*i+j*j != 0)
           z[N+j][N+i] = x[N+j][N+i] / (i*i+j*j);
         else
           z[N+j][N+i] = 0.0;
         }
```

En la optimización, a la posición [N, N] se le asigna el valor “0,0” fuera del bucle, por lo que no se necesitan los condicionales que había antes.

```
for(k=0; k<ITER; k++) {
   for(j=-N; j<N; j++) {
      for(i=-N; i<0; i++)
         z[N+j][N+i] = x[N+j][N+i] / (i*i+j*j);
      for(i=1; i<N; i++)
         z[N+j][N+i] = x[N+j][N+i] / (i*i+j*j);
      }
      z[N][N] = 0.0; }
```

**Importante:** Por error del enunciado, las versiones no hacen exactamente lo mismo. Ya que en la versión no optimizada se accede a todos los elementos de la matriz. Mientras que en la versión optimizada la variable ***‘i’*** nunca llega a valer cero, por lo que nunca se llega a acceder a la columna N de las matrices (excepto para el valor *z[N,N]*, que se modifica fuera del bucle ).

Lo que se puede esperar a priori de la eliminación de los condicionales es que el tiempo por iteración de la versión optimizada sea menor ya que la complejidad de esta última sería de $(2N)^2$ mientras que la de la primera sería $(2N)^2$ x (tiempo del condicional). Aunque sólo con esto no es suficiente para determinar si existe tal mejora, por lo que se tienen que realizar pruebas para ambas versiones y analizar los resultados temporales obtenidos.

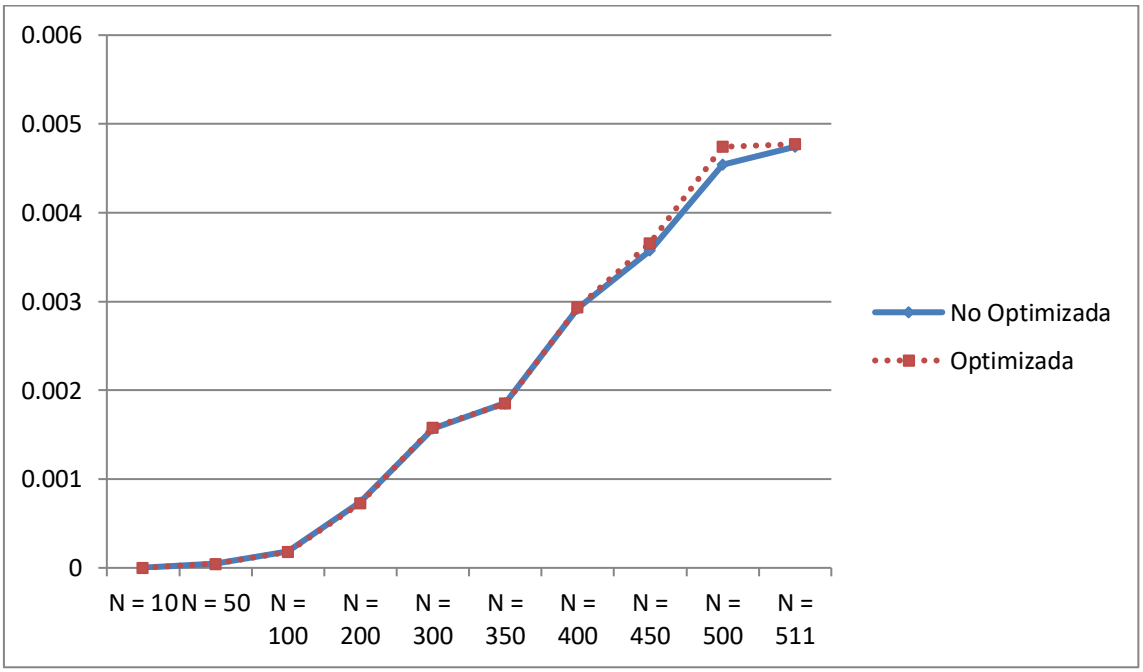
## 3. Ejecución

La ejecución del código se realiza mediante un script en el que hay un bucle que ejecuta el programa compilado tantas veces como diferentes valores de N hayamos declarado. Para luego introducir en una hoja de cálculo los resultados obtenidos. La compilación se realiza con la opción **–O0** para que no se realicen optimizaciones de código.

```
#!/bin/bash

valoresN=(100 250 500 750 1000 3500 7500 10000 11000 12000 13000 14000 15000 16000)
rm resultados.csv
touch resultados.csv
echo -e "Version\tN\tIteracion\tTiempo\tTiempo/Iteracion" >> resultados.csv

gcc -O0 mainDinamico.c

for N in "${valoresN[@]}" ; do
    ./a.out $N >> resultados.csv
done;
```

### 3.1 Estudio preliminar

Una vez realizadas las primeras ejecuciones, se descubre que el tamaño máximo que puede obtener **N** era **N** = 511 y que para los valores permitidos, no se observan diferencias notables en el tiempo por iteración entre las versiones de código.




Por lo que se decide cambiar el código para que las matrices en vez de estáticas sean dinámicas y el tamaño de las matrices pueda ser mayor, siendo el máximo encontrado **N** = 16000, y que esto permitiese ver si para valores mayores se encuentran diferencias en los tiempos de ejecución.

```
float *x = (float *)malloc(2*N*2*N * sizeof(float));
float *z = (float *)malloc(2*N*2*N * sizeof(float));
```

## 3.2 Código ensamblador

Para poder realizar una observación mucho más detallada de la ejecución, se compila el archivo en código C con las opciones:

```
$ gcc -S -fverbose-asm -O0 mainDinamico.c
```

Con la opción **–S** se genera un archivo con el código ensamblador del programa y con la opción **–fverbose-asm** este archivo es mucho más legible al contar con el número de la línea en el programa a la que pertenecen las instrucciones en ensamblador.

A continuación se muestran las instrucciones de la versión no optimizada que se diferencian con la optimizada:

```
.L14:
# mainDinamico.c:35:                for(i=-N; i<N; i++){
    movl    -80(%rbp), %eax # N, tmp292
    negl    %eax    # tmp291
    movl    %eax, -92(%rbp) # tmp291, i
    jmp .L10    #
.L13:
# mainDinamico.c:36:                    if(i*i+j*j != 0)
    movl    -92(%rbp), %eax # i, tmp293
    imull   -92(%rbp), %eax # i, tmp293
    movl    %eax, %edx  # tmp293, _29
    movl    -88(%rbp), %eax # j, tmp294
    imull   -88(%rbp), %eax # j, _30
    addl    %edx, %eax  # _29, _31
    testl   %eax, %eax  # _31
    je  .L11    #,
```

```
.L12:
# mainDinamico.c:35:                for(i=-N; i<N; i++){
    addl    $1, -92(%rbp)   #, i
.L10:
# mainDinamico.c:35:                for(i=-N; i<N; i++){
    movl    -92(%rbp), %eax # i, tmp316
    cmpl    -80(%rbp), %eax # N, tmp316
    jl  .L13    #,
```

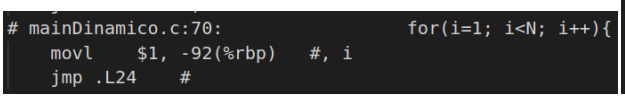
En este caso vemos que las instrucciones del ***for*** que se ejecutan al comienzo del bucle son las de guardar el valor de N, poner ese valor en negativo y guardarlo en ‘i’. A continuación salta a L10, donde comprueba si ‘i’ es igual a N, que en caso negativo salta al ***if*** de L13. Llegados aquí realiza dos operaciones de multiplicación y una de suma con el resultado de cada una, para después comprobar con ***testl*** si el resultado completo es diferente de 0.

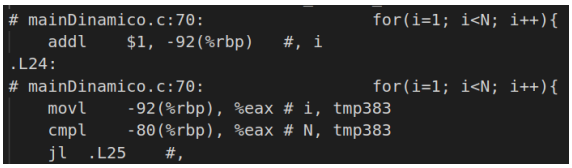
Ahora se muestran las instrucciones de la versión optimizada que se diferencian de las antes vistas:

```
.L26:
# mainDinamico.c:67:                for(i=-N; i<0; i++){
    movl    -80(%rbp), %eax # N, tmp354
    negl    %eax    # tmp353
    movl    %eax, -92(%rbp) # tmp353, i
    jmp .L22    #
```

```
# mainDinamico.c:67:                for(i=-N; i<0; i++){
    addl    $1, -92(%rbp)   #, i
.L22:
# mainDinamico.c:67:                for(i=-N; i<0; i++){
    cmpl    $0, -92(%rbp)   #, i
    js  .L23    #,
```

Se observa que las instrucciones de inicialización son las mismas y que en la comprobación de L22, “i<0”, se ahorra una instrucción de copia ***movl*** ya que no está comparándola con N como antes.




En el segundo bucle ***for*** de la versión optimizada se aprecia que en su inicialización sólo se necesita una instrucción de copia del valor 1 en el registro de la variable 'i', además del salto. Y una vez hecho el salto ahora sí que se vuelve a tener la comprobación "i<N" para la cual se necesita de nuevo la instrucción de copia y la de comparación pertinentes.

Para poder hacerse una idea de los resultados que se van a obtener se puede contar el número de operaciones de cada tipo que se hacen por cada columna recorrida de la matriz.

| Operación | No optimizada | Optimizada |
|---|---|---|
| ***movl*** | 2+8*N ≈ 8*N | 3+N ≈ N |
| ***negl*** | 1 | 1 |
| ***cmpl*** | 2*N | 2*N |
| ***imull*** | 4*N | |
| ***addl*** | 2*N | |
| ***testl*** | 2*N | |

Se aprecia que la diferencia en el número de instrucciones es bastante amplia en favor de la versión optimizada, por lo que se puede esperar que el tiempo por iteración en la segunda versión sea menor que en la primera a medida que va aumentando el valor de **N**.

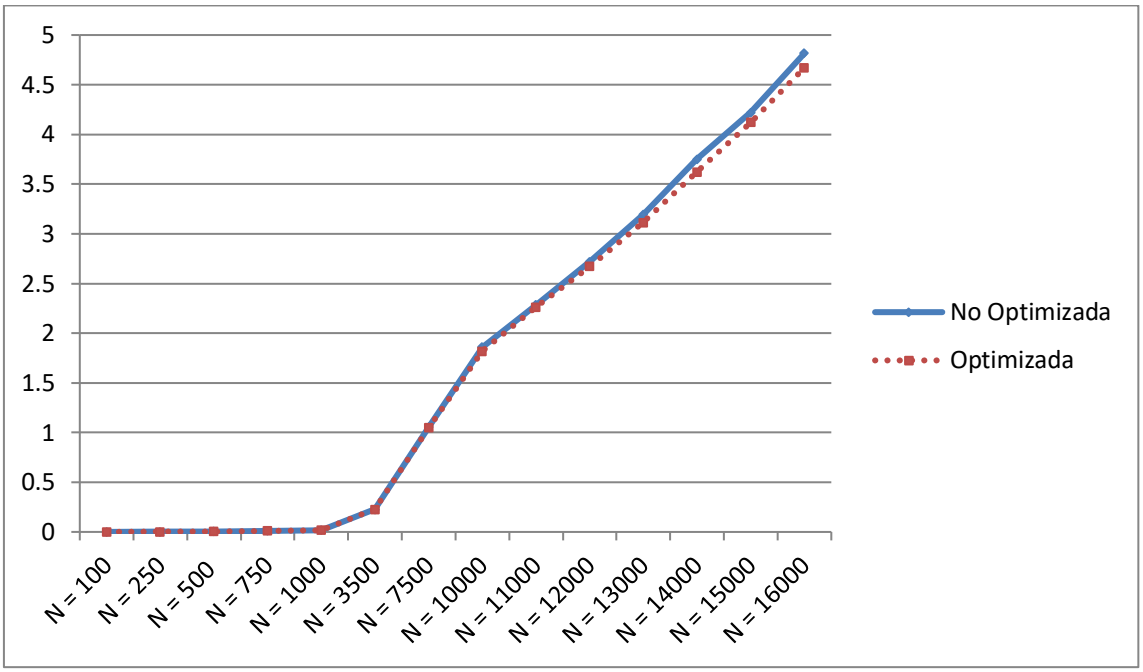
## 3.3 Resultados obtenidos

A pesar de las suposiciones hechas en base al número de instrucciones que se ejecutan, los resultados muestran que no hay diferencias hasta tamaños altísimos de las matrices, y que estas diferencias son de décimas de segundo por iteración.

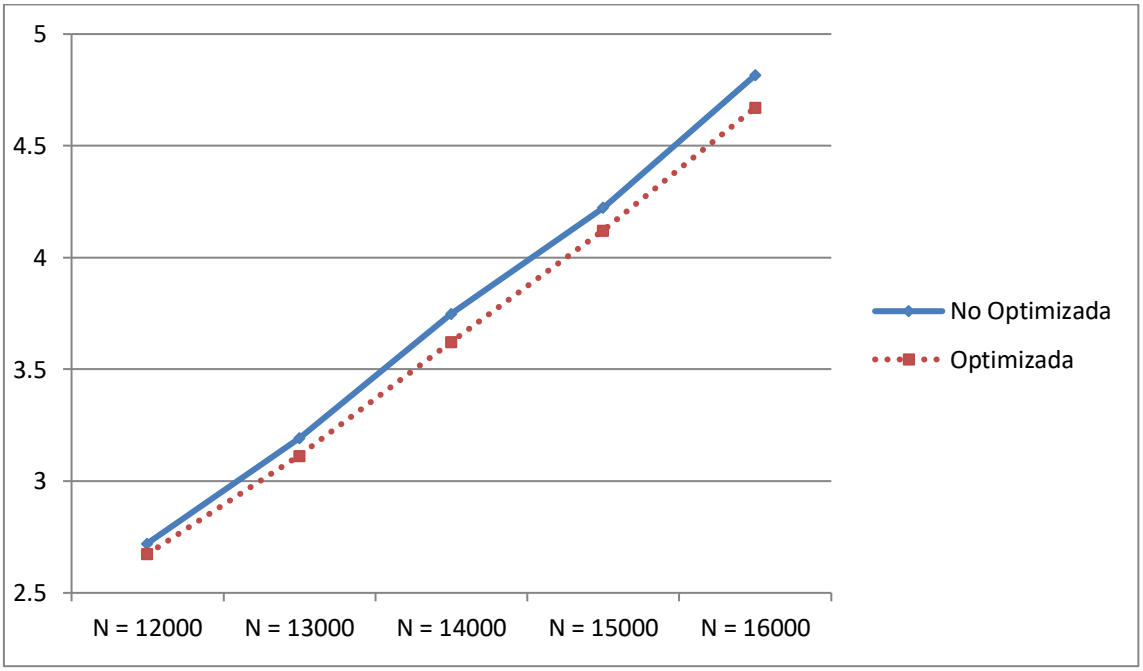
| N | No Optimizada | Optimizada |
|---|---|---|
| 100 | 0,000203 | 0,000192 |
| 250 | 0,001242 | 0,001135 |
| 500 | 0,004685 | 0,004568 |
| 750 | 0,010785 | 0,010426 |
| 1000 | 0,019729 | 0,018721 |
| 3500 | 0,22593 | 0,226506 |
| 7500 | 1,056094 | 1,049813 |
| 10000 | 1,861366 | 1,820323 |
| 11000 | 2,286897 | 2,262062 |
| 12000 | 2,721961 | 2,67378 |
| 13000 | 3,193215 | 3,114291 |
| 14000 | 3,750105 | 3,623359 |
| 15000 | 4,223368 | 4,121831 |
| 16000 | 4,816527 | 4,671284 |

## 4. Análisis

A continuación se representan los datos obtenidos en una gráfica donde el eje Y hace referencia al tiempo de ejecución por iteración.




Como se comenta en el apartado *3.3 Resultado obtenidos*, el tiempo no mejora hasta pasado el tamaño de N = 13.000, aunque sólo lo hace en una o dos décimas de segundo. En la siguiente gráfica se puede ver más detalladamente.




Lo que se puede sacar de esta situación es que lo que se mejora reduciendo el número de instrucciones no compensa el tiempo que suponen los fallos caché que hay en ambas versiones a medida que aumenta el tamaño de N. Esto es bastante lógico ya que el número de

fallos de caché que se darán en una sola fila de la matriz van a ser prácticamente los mismos tanto para la versión optimizada como para la no optimizada.

El único caso en el que se puede notar una diferencia palpable sería al tener unos valores de N muy grandes como los de la última gráfica y que se decida hacer bastantes iteraciones (del orden de $10^2$ en adelante), ya que se ahorrarían bastantes segundos. Aunque esto no afecta a los segundos por iteración, que es lo que se está analizando.

## 5. Conclusiones

Se puede sacar como conclusión general que la técnica de optimización consistente en la minimización de los condicionales realmente funciona. Ahora bien, con los resultados obtenidos se puede determinar que su influencia es muy baja a la hora de conseguir resultados palpables, ya que para notarla, el tamaño del bucle de iteraciones tiene que crecer hasta niveles en los que las variaciones en el tiempo de ejecución van a estar determinadas mayormente por el número de fallos caché y no por el ahorro en el número de instrucciones que supone un condicional.

También se añade que las operaciones realizadas dentro del condicional pueden influir en diferente medida al tiempo de ejecución final, pero recalcando que no será apenas determinante.