Xeración e optimización.

Compiladores e interpretes - Práctica 2

Decembro 2021

Índice

1. Introdución.

Neste informe abordaremos unha técnica de optimización que podería realizar o compilador baseada na localidade. Abordaremos a propia técnica, proporemos probas e avaliaremos o resultado.

2. Técnica analizada.

Neste apartado recollemos as mostras de código empregadas, en que sentido supón unha optimización e que alcance pensamos que podería ter.

2.1. Mostras de código.

O código inicial sería:

```
0  void coidgo()
1  {
2      int i, j;
3
4      typedef struct {float x, y, z;} nuevotipo;
5
6      nuevotipo* v1 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
7      nuevotipo* v2 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
8      nuevotipo* v3 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
9
10     for(j=0; j<ITER; j++){
11         for(i=0; i<N; i++)
12             v3[i].x = v1[i].x + v2[i].x;
13         for(i=0; i<N; i++)
14             v3[i].y = v1[i].y - v2[i].y;
15         for(i=0; i<N; i++)
16             v3[i].z = v1[i].z * v2[i].z;
17     }
18 }
```

O código optimizado sería:

```
0  void coidgo()
1  {
2      int i, j;
3
4      typedef struct {float x, y, z;} nuevotipo;
5
6      nuevotipo* v1 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
7      nuevotipo* v2 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
8      nuevotipo* v3 = (nuevotipo*) malloc(sizeof(nuevotipo)*N);
9
10     for(j=0; j<ITER; j++){
11         for(i=0; i<N; i++) {
12             v3[i].x = v1[i].x + v2[i].x;
13             v3[i].y = v1[i].y - v2[i].y;
14             v3[i].z = v1[i].z * v2[i].z;
15         }
```

```
16      }
17  }
```

Nunha primeira versión de ambas implantacións empregamos memoria estática, pero tras realizar unhas probas inicias consideramos que o comportamento podería variar para tamaños superiores (con memoria estática acadamos `N` máximo para o tamaño dos vectores de 200 000) polo que alteramos a implantación proposta incluíndo memoria dinámica.

2.2. Optimización.

No código anterior constrúense 3 vectores de estruturas de `C`. A estrutura conten 3 campos de tipo `float`. Isto é importante dado que en memoria estarán situados de forma próxima os datos de un mesmo vector e dentro de cada elemento os datos das súas compoñentes.

A optimización proposta cosinte en que en lugar de iterar a través dun único campo por todos os elementos dos vectores, iterase accedendo e operando sobre os tres campos "simultaneamente", por cada iteración do bucle xa que a execución é secuencial.

Deste xeito, pódese empregar o principio de localidade tanto temporal coma espacial, tendo menos fallos cache e precisando realizar un número menor de acceso a memoria. Cando o tamaño do problema é algo grande, sexa polo número de campos da estrutura ou o espazo en memoria que ocupan os campos, esta diferencia irase facendo maior xa que o custo de carga dos datos será maior polo "Memmory Gap". Este efecto xa é apreciable na carga dos distintos niveis de cache. Canto maior sexa a penalización por acceso a memoria máis diferencia haberá nos tempos obtido polo emprego desta técnica.

2.3. Alcance da técnica.

Esta técnica ten como primeira limitación a detección por parte do compilador, xa que resulta difícil que o compilador a puidese aplicar con campos relativamente diversos, ou con lixeiras irregularidades como que non todos os campos fosen usados en todos os vectores.

3. Código empregado e probas.

Para as probas empregamos un ficheiro Python Jupyter Notebook: dispoñible xunta a este documento para que as probas podan ser repetibles, os dous ficheiros de código en `C` da versión inicial e optimizada asi coma o resultado da compilación para obter o código ensamblador de cada versión.

3.1. Sistema de probas.

- **Sistema Operativo:** Ubuntu 20.04 focal.

- **Kernel:** x86_64 Linux 5.11.0-41-generic.
- **CPU:** Intel Core i7-10750H @ 12x 5GHz.
- **Cache L1 Datos:** 192 KiB.
- **Cache L1 Intruccións:** 192 KiB.
- **Cache L2:** 1.5 MiB.
- **Cache L3:** 12 MiB.
- **RAM:** 31 947MiB.

A versión de GCC empregada para todas as compilacións foi `gcc (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0` sempre executada coas opcións `-O0 -static`.

3.2. Ensamblador.

Para poder avaliar o comportamento de ambas versións do código e polo tanto da técnica proposta debemos asegurarnos de que o código executable xerado polo compilador ten a estrutura que nos programamos e o compilador non fixo alteracións significativas que poda afectar ao rendemento respecto das propostas.

Para este obxectivo axudarémonos da ferramenta `Compiler Explorer` e do propio compilador de GCC. A través das dúas ferramentas obteremos o código ensamblador para as nosas propostas de código en C. De xeito que o obtido por polo compilador gcc será estritamente o que executaremos nas nosas probas, mentres que o código xerado por `Compiler Explorer` axudaranos a comprender que se fai en cada parte do código.

Os código obtidos gardáronse en catro ficheiros unha versión por cada ferramenta e código.

- Os resultados de GCC están dispoñibles nos ficheiros: `inicialDin.s` e `optimizadoDin.s`.
- Os resultados de Compiler Explorer están dispoñibles nos ficheiros: `CompExp_inicial.s` e `CompExp_Opt.s`.

No caso de Compiler Explorer escollemos a versión `x86-64 gcc 9.3` para obter os resultados máis semellantes respecto do obtido con GCC, facendo uso das mesmas opcións que no caso de GCC recollidas no apartado anterior deste documento.

3.2.1. Comparativa entre Compiler Explorer e GCC.

Non resulta unha tarefa sinxela nin é obxecto deste informe realizar unha revisión profunda do código ensamblador xerado para todo o programa, en consecuencia poremos o noso foco na tradución feita da función `void codigo()`.

Facendo unha revisión xeral do código apreciamos que entre os resultados xerados por GCC e por Compiler Explorer existen diferencias respecto da instrucións usadas, aínda que non semella ter efecto algún sobre o comportamento do código, exemplos disto son as instrucións `mov` ou `lea` empregadas por Compiler Explorer que no caso de GCC son substitutas por `(movel / movq)` e `leaq`. Se pasmos a función `void codigo()` na saída de Compiler Explorer esta etiquetada como `codigo` e no caso de GCC está como `_Z6codigo`, ambos os códigos teñen lixeiras diferencias pero non atopamos ningunha que poda alterar o seu comportamento.

3.2.2. Análise do código.

Neste apartado traballaremos sobre os códigos obtidos por Compiler Explorer e sobre a rexión da función `void codigo()`. Non existen diferencias entre a versión inicial e a optimizada respecto da reserva de memoria, ocorre o mesmo respecto do primeiro bucle, é hai onde comezan as diferenzas.

Na versión inicial vemos o seguinte fragmento de código onde ser realizan os tres bucles secuencialmente, sinaláronse sobre o propio código.

```
0  .L13:                          <=======COMEZA BUCLE 1
1          mov     DWORD PTR [rbp-4], 0
2          jmp     .L7
3  .L8:
4          mov     eax, DWORD PTR [rbp-4]
5          movsx   rdx, eax
6          mov     rax, rdx
7          add     rax, rax
8          add     rax, rdx
9          sal     rax, 2
10         mov     rdx, rax
11         mov     rax, QWORD PTR [rbp-16]
12         add     rax, rdx
13         movss   xmm1, DWORD PTR [rax]
14         mov     eax, DWORD PTR [rbp-4]
15         movsx   rdx, eax
16         mov     rax, rdx
17         add     rax, rax
18         add     rax, rdx
19         sal     rax, 2
20         mov     rdx, rax
21         mov     rax, QWORD PTR [rbp-24]
22         add     rax, rdx
23         movss   xmm0, DWORD PTR [rax]
24         mov     eax, DWORD PTR [rbp-4]
25         movsx   rdx, eax
26         mov     rax, rdx
27         add     rax, rax
28         add     rax, rdx
29         sal     rax, 2
30         mov     rdx, rax
31         mov     rax, QWORD PTR [rbp-32]
32         add     rax, rdx
33         addss   xmm0, xmm1
34         movss   DWORD PTR [rax], xmm0
```

```
35          add     DWORD PTR [rbp-4], 1
36  .L7:
37          mov     eax, DWORD PTR N[rip]
38          cmp     DWORD PTR [rbp-4], eax
39          jl      .L8
40          mov     DWORD PTR [rbp-4],      <=======COMEZA BUCLE 2
41          jmp     .L9
42  .L10:
43          mov     eax, DWORD PTR [rbp-4]
44          movsx   rdx, eax
45          mov     rax, rdx
46          add     rax, rax
47          add     rax, rdx
48          sal     rax, 2
49          mov     rdx, rax
50          mov     rax, QWORD PTR [rbp-16]
51          add     rax, rdx
52          movss   xmm0, DWORD PTR [rax+4]
53          mov     eax, DWORD PTR [rbp-4]
54          movsx   rdx, eax
55          mov     rax, rdx
56          add     rax, rax
57          add     rax, rdx
58          sal     rax, 2
59          mov     rdx, rax
60          mov     rax, QWORD PTR [rbp-24]
61          add     rax, rdx
62          movss   xmm1, DWORD PTR [rax+4]
63          mov     eax, DWORD PTR [rbp-4]
64          movsx   rdx, eax
65          mov     rax, rdx
66          add     rax, rax
67          add     rax, rdx
68          sal     rax, 2
69          mov     rdx, rax
70          mov     rax, QWORD PTR [rbp-32]
71          add     rax, rdx
72          subss   xmm0, xmm1
73          movss   DWORD PTR [rax+4], xmm0
74          add     DWORD PTR [rbp-4], 1
75  .L9:
76          mov     eax, DWORD PTR N[rip]
77          cmp     DWORD PTR [rbp-4], eax
78          jl      .L10
79          mov     DWORD PTR [rbp-4], 0        <=======COMEZA BUCLE 3
80          jmp     .L11
81  .L12:
82          mov     eax, DWORD PTR [rbp-4]
83          movsx   rdx, eax
84          mov     rax, rdx
85          add     rax, rax
86          add     rax, rdx
87          sal     rax, 2
88          mov     rdx, rax
89          mov     rax, QWORD PTR [rbp-16]
90          add     rax, rdx
91          movss   xmm1, DWORD PTR [rax+8]
```

```
92          mov     eax, DWORD PTR [rbp-4]
93          movsx   rdx, eax
94          mov     rax, rdx
95          add     rax, rax
96          add     rax, rdx
97          sal     rax, 2
98          mov     rdx, rax
99          mov     rax, QWORD PTR [rbp-24]
100         add     rax, rdx
101         movss   xmm0, DWORD PTR [rax+8]
102         mov     eax, DWORD PTR [rbp-4]
103         movsx   rdx, eax
104         mov     rax, rdx
105         add     rax, rax
106         add     rax, rdx
107         sal     rax, 2
108         mov     rdx, rax
109         mov     rax, QWORD PTR [rbp-32]
110         add     rax, rdx
111         mulss   xmm0, xmm1
112         movss   DWORD PTR [rax+8], xmm0
113         add     DWORD PTR [rbp-4], 1
114 .L11:
115         mov     eax, DWORD PTR N[rip]
116         cmp     DWORD PTR [rbp-4], eax
117         jl      .L12
```

A continuación amosase o código da versión optimizada onde temos un único bucle respecto dous que atopábanmos previamente. Ademais do código do bucle sinalamos o comezo das operacións de cada compoñente.

```
0   .L9:                    <═══════ NICO  BUCLE
1           mov     DWORD PTR [rbp-4], 0
2           jmp     .L7
3   .L8:
4           mov     eax, DWORD PTR [rbp-4]      <——— COMPOENTE X
5           movsx   rdx, eax
6           mov     rax, rdx
7           add     rax, rax
8           add     rax, rdx
9           sal     rax, 2
10          mov     rdx, rax
11          mov     rax, QWORD PTR [rbp-16]
12          add     rax, rdx
13          movss   xmm1, DWORD PTR [rax]
14          mov     eax, DWORD PTR [rbp-4]
15          movsx   rdx, eax
16          mov     rax, rdx
17          add     rax, rax
18          add     rax, rdx
19          sal     rax, 2
20          mov     rdx, rax
21          mov     rax, QWORD PTR [rbp-24]
22          add     rax, rdx
23          movss   xmm0, DWORD PTR [rax]
24          mov     eax, DWORD PTR [rbp-4]
```

```
25          movsx   rdx, eax
26          mov     rax, rdx
27          add     rax, rax
28          add     rax, rdx
29          sal     rax, 2
30          mov     rdx, rax
31          mov     rax, QWORD PTR [rbp-32]
32          add     rax, rdx
33          addss   xmm0, xmm1
34          movss   DWORD PTR [rax], xmm0
35          mov     eax, DWORD PTR [rbp-4]      <------- COMPOENTE Y
36          movsx   rdx, eax
37          mov     rax, rdx
38          add     rax, rax
39          add     rax, rdx
40          sal     rax, 2
41          mov     rdx, rax
42          mov     rax, QWORD PTR [rbp-16]
43          add     rax, rdx
44          movss   xmm0, DWORD PTR [rax+4]
45          mov     eax, DWORD PTR [rbp-4]
46          movsx   rdx, eax
47          mov     rax, rdx
48          add     rax, rax
49          add     rax, rdx
50          sal     rax, 2
51          mov     rdx, rax
52          mov     rax, QWORD PTR [rbp-24]
53          add     rax, rdx
54          movss   xmm1, DWORD PTR [rax+4]
55          mov     eax, DWORD PTR [rbp-4]
56          movsx   rdx, eax
57          mov     rax, rdx
58          add     rax, rax
59          add     rax, rdx
60          sal     rax, 2
61          mov     rdx, rax
62          mov     rax, QWORD PTR [rbp-32]
63          add     rax, rdx
64          subss   xmm0, xmm1
65          movss   DWORD PTR [rax+4], xmm0
66          mov     eax, DWORD PTR [rbp-4]      <------- COMPOENTE Z
67          movsx   rdx, eax
68          mov     rax, rdx
69          add     rax, rax
70          add     rax, rdx
71          sal     rax, 2
72          mov     rdx, rax
73          mov     rax, QWORD PTR [rbp-16]
74          add     rax, rdx
75          movss   xmm1, DWORD PTR [rax+8]
76          mov     eax, DWORD PTR [rbp-4]
77          movsx   rdx, eax
78          mov     rax, rdx
79          add     rax, rax
80          add     rax, rdx
81          sal     rax, 2
```

```
82          mov     rdx, rax
83          mov     rax, QWORD PTR [rbp-24]
84          add     rax, rdx
85          movss   xmm0, DWORD PTR [rax+8]
86          mov     eax, DWORD PTR [rbp-4]
87          movsx   rdx, eax
88          mov     rax, rdx
89          add     rax, rax
90          add     rax, rdx
91          sal     rax, 2
92          mov     rdx, rax
93          mov     rax, QWORD PTR [rbp-32]
94          add     rax, rdx
95          mulss   xmm0, xmm1
96          movss   DWORD PTR [rax+8], xmm0
97          add     DWORD PTR [rbp-4], 1        ### PARTE FINAL DO BUCLE
98  .L7:
99          mov     eax, DWORD PTR N[rip]
100         cmp     DWORD PTR [rbp-4], eax              ### PARTE FINAL
        DO BUCLE
101         jl      .L8
```

Conforme ao exposto consideramos que os códigos obtidos son representativos da técnica de optimización proposta e que a compilación realizada non alterou o noso código substancialmente co que son válidos para continuar co nosa análise.

3.3. Medicións de tempo e experimentación.

Neste apartado recollemos varias apreciacións do xeito realizaremos a medición de tempos e a experimentación. Convén comezar destacando que o obxectivo deste análise non é obter resultados cuantitativos respecto dos tempos, senón servirnos dos resultados temporais para avaliar a posible mellora da aplicación desta técnica, en que casos se da e en que medida.

Conforme ao exposto a medida do tempo realizarémola sobre toda a función que conten os códigos presentados anteriormente, xa que simplifica o código e non supón un problema ao non buscar unha interpretación cuantitativa dos resultados. Se se quixese facer dita interpretación deberiamos ter en conta que estamos incluíndo na medida o tempo de chamada á función e a reserva de memoria ademais do tempo do propio código. Nesta liña tamén tería un impacto o *quecemento cache* que analizaremos con algo máis de coidado agora.

3.3.1. *Quecemento cache.*

Este é un dos aspectos recollidos anteriormente que pode ter impacto nos resultados, pero podémolo mitigar aumentando o número de iteracións que facemos, por isto fixemos unha pequena análise para un valor de N = 5 000.

Vemos como a medida que aumentamos o número de iteracións este valor diminúe ata chegar unha tendencia constante, o que era de esperar. Tomaremos

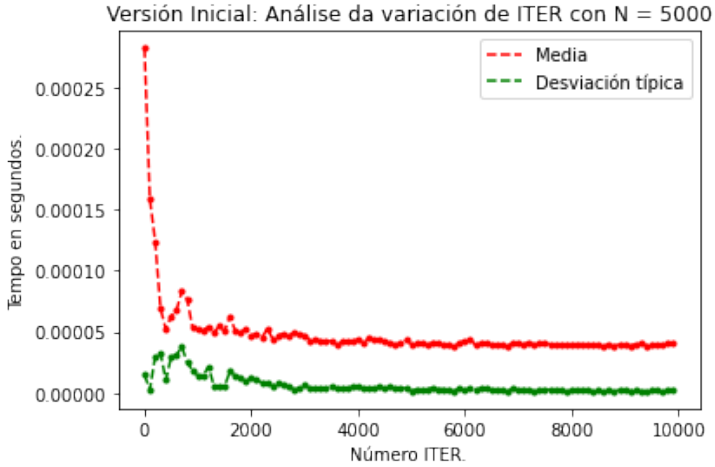

Figura 1:

un valor de 2 000 iteracións como referencia xa que se sitúa preto do valor a partir do cal é constante. Con todo, convén salientar que este efecto será o mesmo para ambas as versións xa que as executaremos completamente por separado.

4. Probas e resultados.

Para a realización das probas empregamos unha `Jupiter Notebook` que estará dispoñible tanto en formato `.ipynb` e `.html` xunto a este informe.

O noso enfoque foi tratar de ter a imaxe máis xeral posible dos resultados de ambas as versións para distintos tamaños, extraer conclusións e fixarnos en aqueles puntos que consideramos para afondar na análise. Destacar que os valores empregados non foron arbitrarios senon que son froito dunha serie de ensaios.

Comezamos avaliando o impacto do *quecemento cache* como xa expuxemos na sección anterior deste documento e unha vez escollido un valor de compromiso para o número de iteracións fixámolo a este (2 000 iteracións). O tempo que a nosa implementación tarde en realizar estas iteracións será o valor devolto polo programa, que para ser interpretado devidirémolo entre o número de iteracións que estamos a realizar. O parámetro N será o foco da análise seguinte.

Dado que os tempos que manexamos pódense ver alterados por elementos alleos ao estudo da técnica coma interrupcións do sistema operativo, que alteren magnificamente as medidas pequenas ou o propio rendemento xeral do sistema por outros procesos, realizaremos varias medidas para cada tamaño escollido. Para este caso realizaremos 10 medidas das que obteremos a súa media e desviación típica como elementos de estudo, aínda que somos conscientes de que son

insuficientes para un estudo estatístico rigoroso, consideramos que nos permitirán ver con suficiente detalle o ocorrido a par que podemos realizar un número suficiente de probas nun tempo razoable.

4.1. Visión xeral.

Para ter unha visión xeral consideramos axeitado ver a evolución dos tempos de con valores de N entre 1 e 1 000 000 cun paso de 100 000. Así veremos se existen cambios nas tendencias a medida que aumentamos o tamaño do problema.

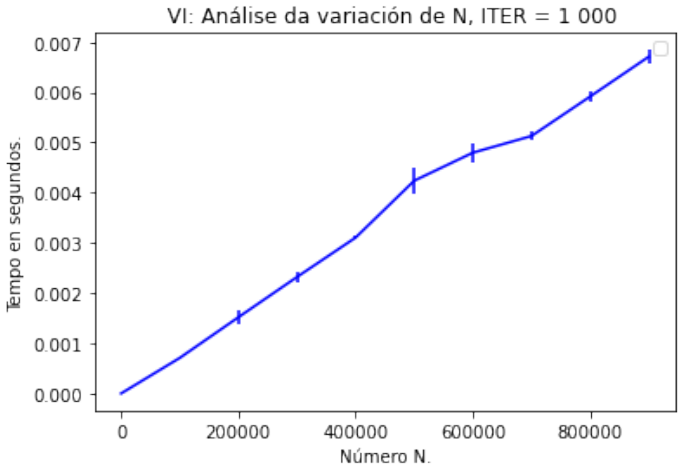

Figura 2:

Na gráfica 4 podemos ver como o tempo ascende a medida que aumentamos o tamaño, o que é de esperar. Ademais podemos ver como a partir do tamaño 500 000 se produce unha subida abrupta e o pendente dos puntos aumenta a partir dese punto, estamos a ver o "Memmory Gap". Isto podémolo comprobar tendo en conta o tamaño da memoria cache de nivel 3 da máquina é de 12 MiB e que os `double` ocupan 8 Bytes. Dividindo o tamaño da cache e recordando que temos e 3 vectores, teremos unha estimación do valor de N a partir do que estariamos empregando de seguro a memoria RAM.

$CacheL3/(sizeof(double) * 3) = 524288$

Moi probablemente estariamos facendo antes deste valor xa que debemos manter máis información na cache e non entraremos en máis profundidade neste aspecto. Este efecto non é apreciable no caso da versión optimizada probablemente porque ao aproveitar a localidade temporal non se acentúan tanto estas diferenzas.

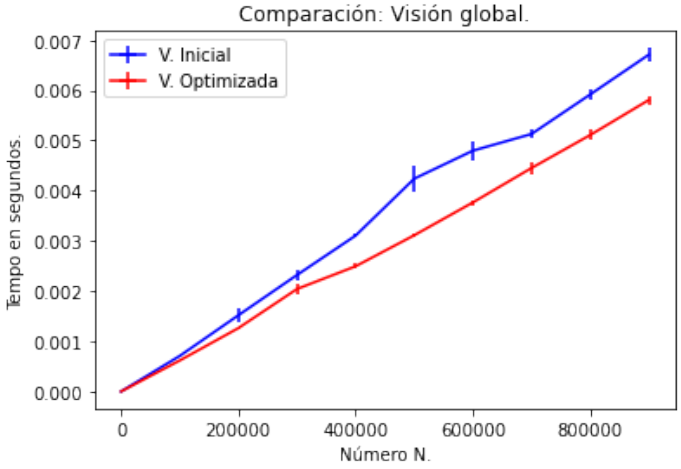

Figura 3:

Apreciamos unha certa tendencia a distanciarse, o que podemos ver na gráfica 3, entre ambas as versións, sendo sempre superiores os tempos da versión inicial respecto da versión optimizada.

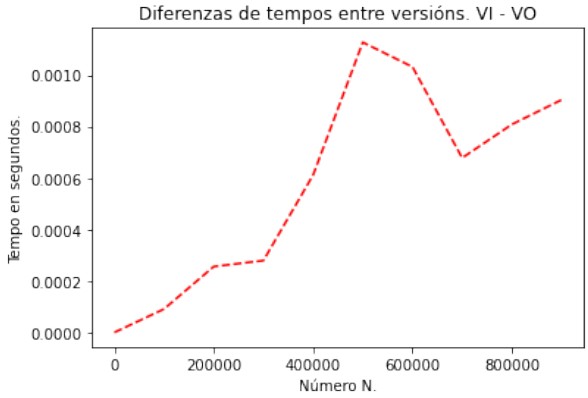

Figura 4:

Por último, se calculamos a media de todos os tempos obtidos polas dúas versións podemos comprobar como a versión optimizada precisa algo máis do 83 % do tempo respecto da versión inicial, supoñendo un importante aforro.

4.2. Tamaños pequenos.

Agora enfocámonos na parte inicial das gráficas onde ambas parecen solaparse.

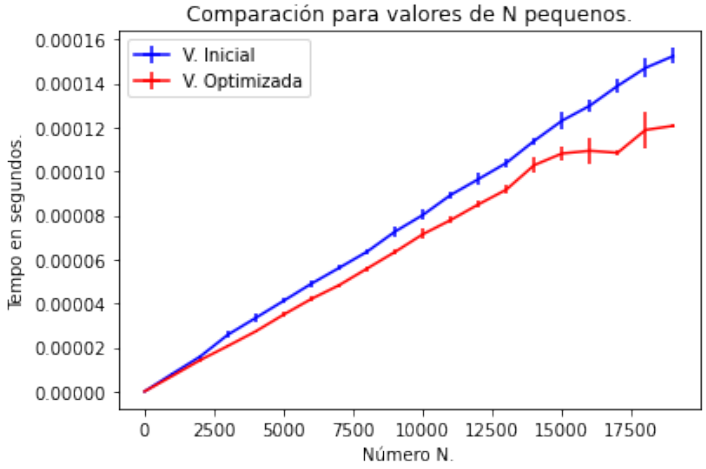

Figura 5:

Na figura 5 podemos apreciar como se mantén o comportamento visto previamente neste caso escalado aos tamaños nos que nos atopamos.

De igual modo que no caso anterior se avaliamos a diferencias de tempos, figura 6, vemos que en ningún caso e menor o tempo da versión inicial. Repetindo de igual modo a proporción entre as medias de tempos volvemos a obter que a versión optimizada precisou en media un 84 % do tempo respecto de versión inicial.

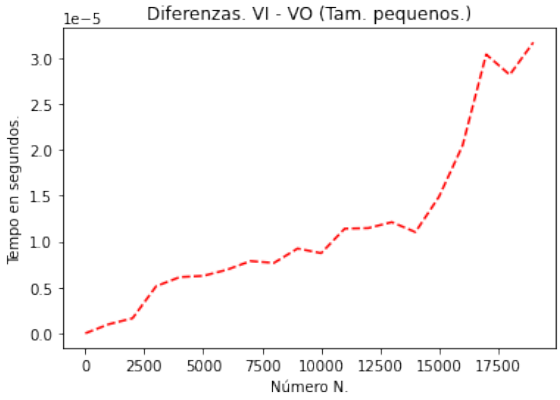

Figura 6:

5. Conclusións.

Para valores pequenos a mellora non semella de importancia, non en tanto, a aplicación da técnica vimos como sempre nos levaba a mellorar ou igualar o tempo preciso co que debemos aplicala sempre que sexa posible.

O principal obstáculo da mesma reside na peculiaridade da situación na que se pode empregar e na complexidade que pode supoñer para o compilador detectar estas circunstancias.

Por último, convén destacar que é unha optimización intuitiva e facilmente aplicable polo programador ao momento de escribir o código a diferencia de outras que poderían resultar máis complicadas e afectarían a lexibilidade do mesmo.