COMPILADORES E INTÉRPRETES

Generación y optimización de código

Práctica 1 - Compiladores

Diego Reiriz Cores - diego.reiriz@rai.usc.es

COMPILADORES E INTÉRPRETES **1**
Parte 1 - Apartado 2.h de la práctica 2
Comparación del tamaño de los códigos objeto 2
Comparación de los tiempos de ejecución 3
Comparación de código ensamblador entre las diferentes versiones: 4
Parte 2 - Apartado 3 de la práctica 5
Uso de –funroll-loops 5
Comparación del código ensamblador 5
Comparación del rendimiento 10

Parte 1 - Apartado 2.h de la práctica

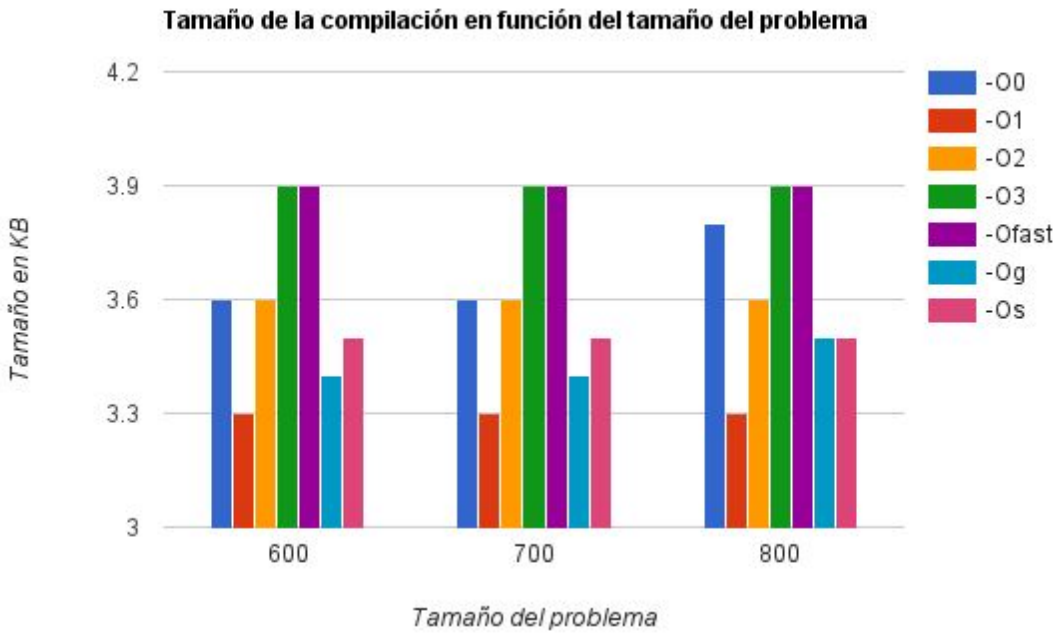
Comparación del tamaño de los códigos objeto

En este apartado se ha compilado el código propuesto en la práctica para diferentes tamaño de problema y utilizando diferentes opciones de compilación.

Las opciones de compilación que se han utilizado han sido las siguientes:

- **O0:** optimización por defecto
- **O1:** realiza pequeñas optimizaciones centradas en reducir el tamaño y el tiempo de ejecución del código.
- **O2:** se basa en el nivel anterior y a mayores realizan más optimizaciones, las cuales intentan obtener rendimiento sin consumir demasiado espacio.
- **O3:** realiza aún más optimizaciones que los casos anteriores a costa de ocupar más espacio.
- **Ofast:** igual que O3, pero también activa algunas optimizaciones no disponibles para todos los compiladores
- **Og:** activa optimizaciones que no dificultan el proceso de depuración
- **Os:** activa todas las opciones de optimización de O2 que no suelen aumentar el tamaño del código y realiza optimizaciones centradas en disminuir el tamaño del código.

A continuación se muestra una gráfica con los resultados obtenidos usando las diferentes opciones de compilación, para tamaños del problema de 600,700 y 800

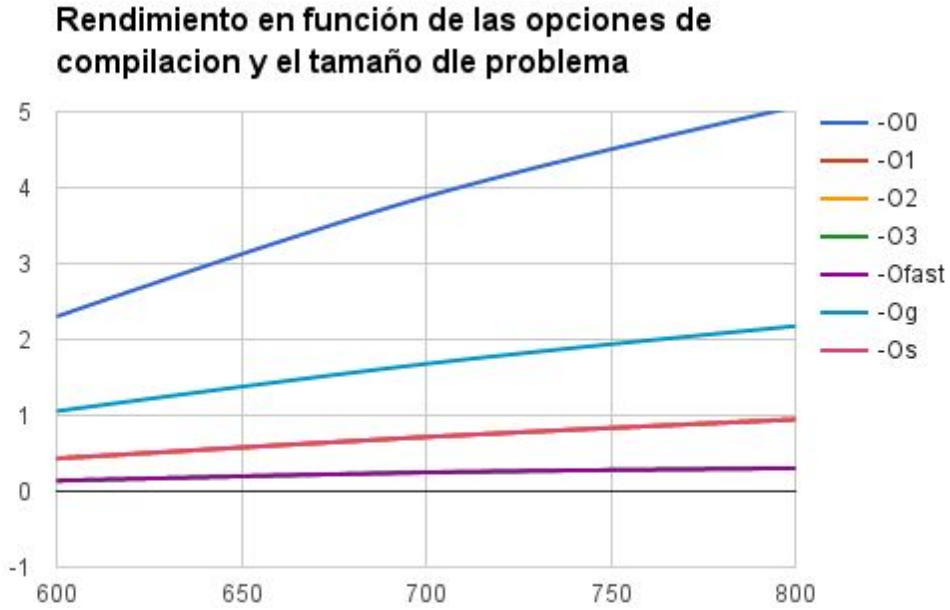


En esta gráfica es interesante destacar:

- el código generado mediante las opciones **-O1** y **-O2** ocupan el mismo o menos espacio que el código sin compilar,
- usando **-O3** obtenemos mejoras de rendimiento(se muestran en el siguiente apartado), pero el código ocupa más que la versión sin optimizar.
- el tamaño del problema puede afectar a la cantidad de código generado por gcc, como se puede observar en la gráfica para un tamaño de problema de **800** y con las opciones de compilación de **-O0** y **-Os**

Comparación de los tiempos de ejecución

El propçosito de este aparato es analizar el impacto en el rendimiento de las diferentes opciones de compilación de gcc. Para ello, se vuelven a utilizar las mismas opciones de compilación que en el apartado anterior, al igual que los valores del tamaño del problema.



NOTA: en la gráfica -O1, -O2 y -Os se solapan, al igual que -O3 y -Ofast

En la gráfica podemos ver que en general, el tiempo de ejecución de los códigos generados cumple con las promesas de las diferentes opciones de compilación.

Destacar que en este caso la opción **-O2**, no ha conseguido ninguna mejora de rendimiento apreciable tras aplicar varias técnicas de compilación que han aumentado el tamaño del código (como se ha visto en el apartado anterior). Por lo que en este caso carecería de sentido usar esta opción en un caso real.

En el caso de la opción **-Os**, esta tampoco consigue ninguna mejora de rendimiento apreciable frente a **-O1** o **-O2**, pero el código generado por esta es más compacto que el generado por **-O2.** Al igual que en la comparación realizada de -O2 con -O1, sigue sin ser

rentable la utilización de esta opción en este caso, ya que el código ocupa más espacio pero no nos proporciona ninguna mejora.

Comparación de código ensamblador entre las diferentes versiones:

En este apartado se procederá a la comparación de los diferentes versiones de código generadas mediante el compilador gcc, utilizando diferentes opciones de compilación.

Primero empezaremos comparando la versión generada sin optimizar frente a la versión generada utilizando la flag **-O1,** sobre la cual se han aplicado opciones básicas de optimización.

La primer diferencia que se puede apreciar entre ambos archivos, es que la versión generada utilizando la opción **-O1**, no tiene al inicio del código ensamblador las declaraciones de variables y estructuras que se utilizan en el programa.

Así mismo podemos observar como el código generado por la opción **-O1** es más compacto que el generado por la opción **-O0**. Esto en parte se debe a la utilización de más registros en la versión optimizada. Concretamente podemos ver como los registros xmm, los cuales están pensados para la realización de operaciones vectoriales, son usados desde el 1 al 8, mientras que en la versión sin optimizar solo se usan del 0 al 2.

Por último ahora también se puede apreciar como se han restructurado ciertos fragmentos de código en la versión optimizada. Por ejemplo, el ćodigo que se encarga de modificar el tamaño de la pila, ahora se encuentra agrupado principalmente bajo la etiqueta L10, aunque también aparece duplicado una única vez al inicio de la función main del programa.

Ahora procederemos a comparar la versión optimizada mediante la opción -O2 con la opción optimizada mediante **-O1**.

En este caso los cambios principales que se pueden observar en el código se deben a la utilización de pseudo-operaciones por parte del compilador, en vez de utilizar directamente múltiples llamadas a funciones en código ensamblador. Algunos ejemplos de estas pseudo-operaciones son el uso de:

- **.palign:** se usa principalmente para realizar el cálculo de potencias de 2.
- **.cfi_remember_state y .cfi_restore_state**: no he sido capaz de encontrar una definición clara de la labor de estas pseudo-operaciones. Se puede consultar algo de información en el siguiente enlace http://stackoverflow.com/questions/14582306/implementation-of-cfi-remember-state

A continuación se muestran las diferencia encontradas del código compilado utilizando la opción -O3 frente al compilado usando la opción -O2. Estas básicamente radican en que el código optimizado mediante la opción -O3 usa operaciones vectoriales para manipular

varios registros al mismo tiempo, lo cual no había hecho ninguna opción de optimización previamente. Algunos ejemplos de estas operaciones vectoriales son:

- paddd
- pshufd

Así mismo también se puede observar como se produce cierto alineamiento de estructuras de datos en memoria al final del código generado mediante la opción **-O3**, lo que tiene bastante sentido teniendo en cuenta que ha pasado a aplicar operaciones vectoriales en algunos casos.

Por último se procederá a comprar el código generado mediante la opción -O2 con el generado por la opción Os, ya que este último realiza las misma optimizaciones que la opción -O2, pero a mayores ejecuta alguna optimización enfocada a evitar un consumo excesivo de memoria.

En este caso, el código generado por ambas opciones es prácticamente igual. La mayor diferencia que se puede apreciar es la substitución de la instrucción **movlhps** por un bloque de código ensamblador de tamaño considerable.

De hecho esta última opción está enfocada para reducir el tamaño del código resultante, pero si contamos el número de líneas de código ensamblador generado en ambos casos, podemos ver como simplemente ocupa 5 líneas menos de ćodigo ensamblador que la opción compilada con -O2

Parte 2 - Apartado 3 de la práctica

Uso de –funroll-loops

Esta opción de gcc le indica al compilador que desenrolle los bucles que se encuentre en el código a compilar de forma que en una iteración del bucle desenrollado se produzca varias iteraciones del bucle sin ser desenrollado. Esta técnica intenta aprovechar la localidad espacial de los datos, ya que si por ejemplo estamos recorriendo un vector de elementos, lo más probable es que cuando se quiera acceder por primera vez a un elemento en el bucle, se produzca un fallo y el sistema traerá de memoria el elemento que ha producido el fallo más los n siguientes (en función del tamaño de los elementos del vector). Por lo tanto, al desenrollar el bucle no estamos aprovechando de la localidad espacial de los datos.

Comparación del código ensamblador

A continuación se procederá a comparar los códigos en ensamblador para comprobar la explicación realizada anteriormente.

Comparación del bucle de inicialización

```
for(i=0;i<N;i++)
        res[i]=0.0005*i;
```

sin -funroll-loops	con -funroll-loops
...	...

.L2:
  pxor %xmm0, %xmm0
  cvtsi2sd %eax, %xmm0
  mulsd %xmm1, %xmm0
  movsd %xmm0, res(,%rax,8)
  addq $1, %rax
 	
  cmpq $10000, %rax
  jne .L2
	.L2:
	  pxor %xmm1, %xmm1
	  cvtsi2sd %eax, %xmm1
	  mulsd %xmm0, %xmm1
	  movsd %xmm1, res(,%rax,8)
	  addq $1, %rax
	  pxor %xmm2, %xmm2
	  cvtsi2sd %eax, %xmm2
	  mulsd %xmm0, %xmm2
	  movsd %xmm2, res(,%rax,8)
	  leaq 1(%rax), %rcx
	  pxor %xmm3, %xmm3
	  cvtsi2sd %ecx, %xmm3
	  mulsd %xmm0, %xmm3
	  movsd %xmm3, res(,%rcx,8)
	  leaq 2(%rax), %rsi
	  pxor %xmm4, %xmm4
	  cvtsi2sd %esi, %xmm4
	  mulsd %xmm0, %xmm4
	  movsd %xmm4, res(,%rsi,8)
	  leaq 3(%rax), %rdi
	  pxor %xmm5, %xmm5
	  cvtsi2sd %edi, %xmm5
	  mulsd %xmm0, %xmm5
	  movsd %xmm5, res(,%rdi,8)
	  leaq 4(%rax), %r8
	  pxor %xmm6, %xmm6
	  cvtsi2sd %r8d, %xmm6
	  mulsd %xmm0, %xmm6
	  movsd %xmm6, res(,%r8,8)
	  leaq 5(%rax), %r9
	  pxor %xmm7, %xmm7
	  cvtsi2sd %r9d, %xmm7
	  mulsd %xmm0, %xmm7
	  movsd %xmm7, res(,%r9,8)
	  leaq 6(%rax), %r10
	  pxor %xmm8, %xmm8
	  cvtsi2sd %r10d, %xmm8
	  mulsd %xmm0, %xmm8
	  movsd %xmm8, res(,%r10,8)
	  addq $7, %rax

...	

	cmpq $10000, %rax jne .L2 ...

En la tabla podemos apreciar como el bucle de inicialización de ha desenrollado sin problemas y ahora se realizan 8 operaciones por iteración frente a la única que se realiza en la versión sin optimizar.

El hecho de que se realicen 8 operaciones al desenrollar el bucle, tiene bastante sentido si nos damos cuenta de que un double en C ocupa 8 bytes en memoria y el tamaño de las líneas caché del procesador en el que se realizaron las mediciones es de 64 bytes, lo que significa que en una línea caché podemos almacenar 8 doubles. Por lo tanto, es posible que en cada operación del bucle se produzca un fallo caché al acceder al primer double (esto depende de la cantidad de prefetching realizada por el procesador), pero el compilador se asegura de que los 7 elementos siguientes se van a encontrar en caché y por lo tanto va a poder acceder a ellos rápidamente.

Comparación del segundo bucle

```
for(i=0;i<N;i++) {
        x=res[i];
        if (x<10.0e6) x=x*x+0.0005;
        else x=x-1000;
        res[i]+=x;
}
```

sin -funroll-loops	con -funroll-loops

...
 	
.L6:
movq %rax, %rdx
movsd (%rax), %xmm0
ucomisd %xmm0, %xmm2
jbe .L10
movapd %xmm0, %xmm1
mulsd %xmm0, %xmm1
addsd %xmm3, %xmm1
jmp .L5
.L10:
movapd %xmm0, %xmm1
subsd %xmm4, %xmm1
.L5:
addsd %xmm1, %xmm0
movsd %xmm0, (%rdx)
addq $8, %rax
cmpq %rcx, %rax
jne .L6
	...
	
	.L6:
	movq %rdx, %rax
	movsd (%rdx), %xmm12
	ucomisd %xmm12, %xmm9
	jbe .L31
	movapd %xmm12, %xmm13
	mulsd %xmm12, %xmm13
	addsd %xmm11, %xmm13
	jmp .L5
	.L31:
	movapd %xmm12, %xmm13
	subsd %xmm10, %xmm13
	.L5:
	addsd %xmm13, %xmm12
	movsd %xmm12, (%rax)
	leaq 8(%rdx), %rcx
	movsd 8(%rdx), %xmm14
	ucomisd %xmm14, %xmm9

. . .
 	
...
	ja .L10
	jmp .L32
	.L40:
	...
	código al final del programa
	...
	.L32:
	movapd %xmm14, %xmm15
	subsd %xmm10, %xmm15
	jmp .L33
	.L10:
	movapd %xmm14, %xmm15
	mulsd %xmm14, %xmm15
	addsd %xmm11, %xmm15
	.L33:
	addsd %xmm15, %xmm14
	movsd%xmm14, (%rcx)
	movsd8(%rcx), %xmm0
	ucomisd %xmm0, %xmm9
	ja .L12
	movapd %xmm0, %xmm1
	subsd %xmm10, %xmm1
	jmp .L34
	.L12:
	movapd %xmm0, %xmm1
	mulsd %xmm0, %xmm1
	addsd %xmm11, %xmm1
	.L34:
	addsd %xmm1, %xmm0
	movsd %xmm0, 8(%rcx)
	movsd 16(%rcx), %xmm2
	ucomisd %xmm2, %xmm9
	ja .L14
	movapd %xmm2, %xmm3
	subsd %xmm10, %xmm3
	jmp .L35
	.L14:
	movapd %xmm2, %xmm3
	mulsd %xmm2, %xmm3
	addsd %xmm11, %xmm3
	.L35:
	addsd %xmm3, %xmm2
	movsd%xmm2, 16(%rcx)
	movsd24(%rcx), %xmm5
	ucomisd %xmm5, %xmm9
	ja .L16
	movapd %xmm5, %xmm4
	subsd %xmm10, %xmm4
	jmp .L36
	.L16:

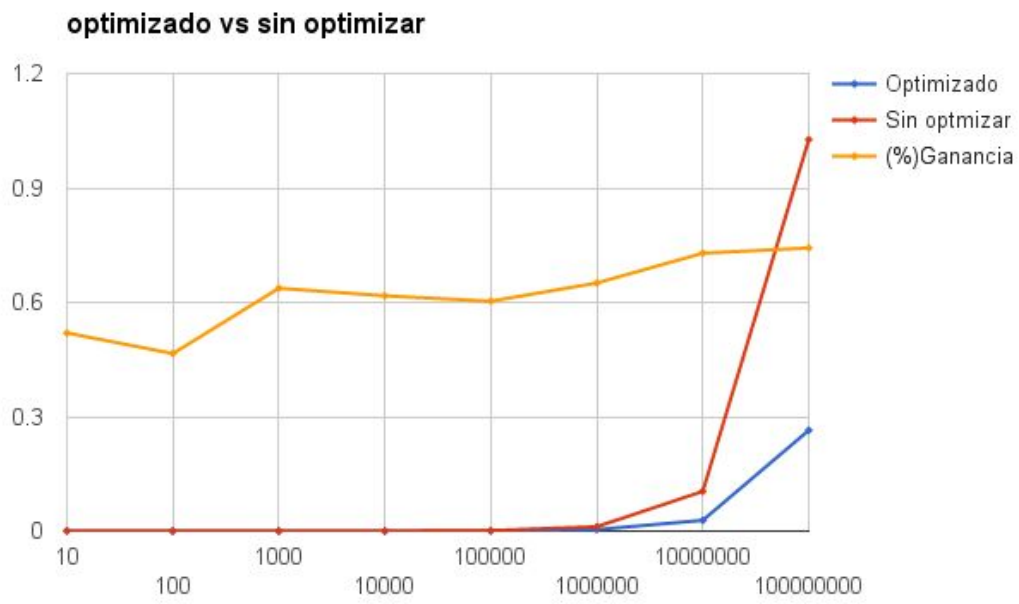
```
        movapd          %xmm5, %xmm4
        mulsd   %xmm5, %xmm4
        addsd   %xmm11, %xmm4
.L36:
        addsd   %xmm4, %xmm5
        movsd %xmm5, 24(%rcx)
        movsd 32(%rcx), %xmm6
        ucomisd         %xmm6, %xmm9
        ja      .L18
        movapd          %xmm6, %xmm7
        subsd   %xmm10, %xmm7
        jmp     .L37
.L18:
        movapd          %xmm6, %xmm7
        mulsd %xmm6, %xmm7
        addsd %xmm11, %xmm7
.L37:
        addsd %xmm7, %xmm6
        movsd%xmm6, 32(%rcx)
        movsd40(%rcx), %xmm8
        ucomisd         %xmm8, %xmm9
        ja      .L20
        movapd          %xmm8, %xmm12
        subsd %xmm10, %xmm12
        jmp     .L38
.L20:
        movapd          %xmm8, %xmm12
        mulsd   %xmm8, %xmm12
        addsd   %xmm11, %xmm12
.L38:
        addsd   %xmm12, %xmm8
        movsd %xmm8, 40(%rcx)
        movsd 48(%rcx), %xmm13
        ucomisd         %xmm13, %xmm9
        ja      .L22
        movapd          %xmm13, %xmm14
        subsd   %xmm10, %xmm14
        jmp     .L39
.L22:
        movapd          %xmm13, %xmm14
        mulsd %xmm13, %xmm14
        addsd %xmm11, %xmm14
.L39:
        addsd %xmm14, %xmm13
        movsd%xmm13, 48(%rcx)
        leaq    56(%rcx), %rdx
        cmpq    %r11, %rdx
        jne     .L6
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
```

	jmp .L40 . . .

Aunque en este caso es un poco más difícil identificar las diferentes iteraciones desenrolladas del bucle, se puede apreciar como este se ha desarrollado correctamente e incluso se repiten las instrucciones **if** y **else** que realiza el bucle para cada elemento, provocando esto que aumente en gran cantidad el ćodigo generado.

A modo de curiosidad, se puede observar como la versión sin el desenrollado de bucles ocupa 1.3KB ,mientras que la versión desenrollada ocupa 3.7K

Comparación del rendimiento



En este apartado se procederá a comparar el rendimiento del código analizado anteriormente, utilizando diferentes tamaño de problema. En concreto, se han utilizado para la comparación los siguientes valores de N: 10^3,10^4,10^5,10^6,10^7,10^8. Se intentó utilizar valores de N superiores, pero al hacerlo se producían errores de ejecución

Básicamente, si observamos los resultados obtenidos en la gráfica podemos observar como la versión optimizada siempre se ejecuta más rápido que la sin optimizar y que la diferencia entre el tiempo de ejecución de ambas versiones aumenta a medida que aumenta el tamaño del problema.

Por último, comentar que para tamaños pequeños del problema, la versión optimizada obtiene sobre un 50% de mejoría en cuanto a coste temporal en el caso de N=10. Este factor de Ganancia o mejoría, también aumenta a medida que aumenta el tamaño del problema, llegando a alcanzar un valor de entorno al 70% en nuestro caso.

Esta gran cantidad de mejoría se debe a que el código principal del programa son 2 lazos que el compilador está optimizando sin problemas, por lo que a medida que aumenta el tamaño de los lazos, pesa menos sobre el total de la ejecución el resto del código del programa.