Compiladores e Interpretes

Generación y Optimización de Código

Práctica 1

Martín Suárez García martin.suarez.garcia@rai.usc.es

21 de diciembre de 2021

Contenido

Compilación con grupos de optimizaciones **1**

Tamaño del código objeto 1

Tiempos de ejecución 2

Código ensamblador 4

Traducción del compilador 4

MIPS64 según Compiler Explorer 4

Discrepancias entre traducciones 5

Conclusiones 5

Compilación con -funroll-loops **6**

Tiempos de ejecución variando N 6

Código ensamblador 8

Traducción del compilador 8

Conclusiones 8

Bibliografía **10**

Anexo - Máquina de pruebas **11**

Compilación con grupos de optimizaciones

En este apartado compilaremos el código con las distintas opciones de optimización (-O1, -O2, -O3 y -Os) y la opción de no optimizar (-O0). Realizaremos una serie de mediciones y comparaciones sobre las distintas opciones.

```c
#include <stdio.h>
#define Nmax 600
void producto(float x, float y, float* z) {
    *z = x * y;
}
main() {
    float A[Nmax][Nmax], B[Nmax][Nmax], C[Nmax][Nmax], t, r;
    int i, j, k;
    for (i = 0; i < Nmax; i++) /* Valores de las matrices */
        for (j = 0; j < Nmax; j++) {
            A[i][j] = (i + j) / (j + 1.1);
            B[i][j] = (i - j) / (j + 2.1);
        }
    for (i = 0; i < Nmax; i++) /* Producto matricial */
        for (j = 0; j < Nmax; j++) {
            t = 0;
            for (k = 0; k < Nmax; k++) {
                producto(A[i][k], B[k][j], &r);
                t += r;
            }
            C[i][j] = t;
        }
}
```

Código 1 - Código base para medir el rendimiento de las opciones de optimización generales

Las características de la máquina empleada para realizar las pruebas se pueden consultar en el apartado Anexo - Máquina de pruebas.

Tamaño del código objeto

Las distintas opciones de optimización traen consigo un efecto sobre el tamaño del programa, pues todas ellas optimizan también el tamaño del programa, intentando reducir su peso (excepto O0, la cual no hace optimización alguna). En la siguiente tabla se recogen los tamaños de los códigos objeto asociados a las distintas opciones.

OPCIÓN DE COMPILACIÓN	TAMAÑO DEL CODIGO OBJETO
-O0	3.7 KB
-O1	3.3 KB
-O2	3.4 KB
-O3	3.4 KB
-Os	3.3 KB

Observamos que el código sin optimizar es el más pesado, mientras que el resto de códigos tienen un tamaño similar, presentando O1 y Os el menor tamaño.

Estos resultados son entendibles, pues O0 no realiza ningún tipo de optimización, mientras que el resto de opciones de compilación realizan optimizaciones para reducir el tamaño del código. No obstante, O2 y O3, al introducir una mayor cantidad de código para realizar optimizaciones de rendimiento, pierden un poco de la mejora de tamaño a cambio de mayor velocidad de ejecución.

Tiempos de ejecución

Para medir los tiempos de ejecución hemos realizado un total de 50 ejecuciones con cada programa compilado siendo el valor de Nmax el establecido en el código inicial. Para los resultados hemos calculado el tiempo máximo, mínimo, media geométrica y mediana. No se han detectado datos outliers, por lo que no ha sido necesario tratarlos de antemano.

En la siguiente gráfica se muestran los resultados obtenidos, representando los tiempos en segundos.

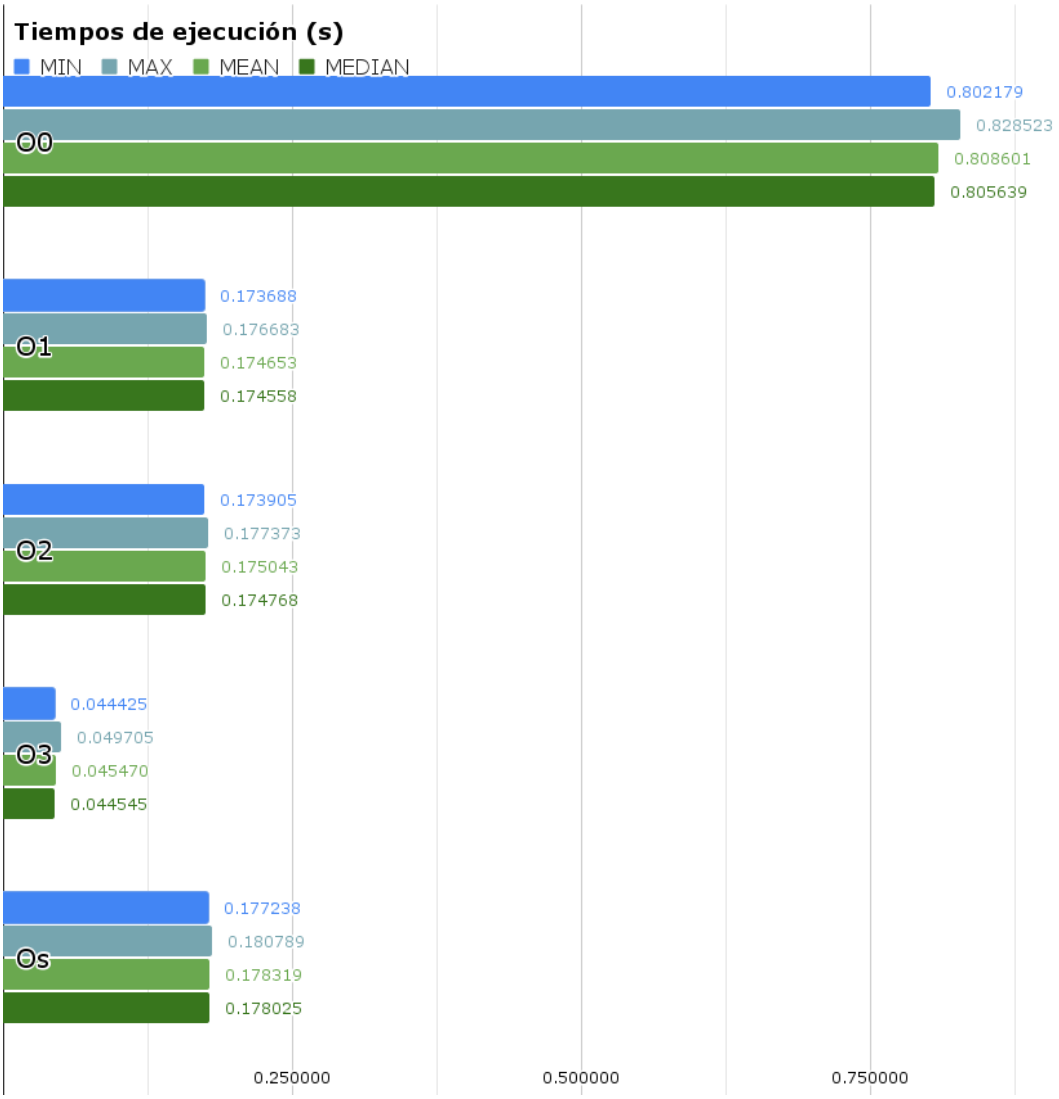

Figura 1 - Tiempos de ejecución (s) de los programas compilados

Se puede observar como el rendimiento del código sin optimizar es pésimo comparado con el resto, siendo este resultado esperable.

Si nos adentramos en las opciones de optimización, podemos observar que O1, O2 y Os presentan tiempos de ejecución parejos, aunque los tiempos de Os son un poco más elevados (aproximadamente 5 milisegundos), debido a que esta opción implementa optimizaciones orientadas a reducir el tamaño del ejecutable y no centrandose en la velocidad de ejecución. Las diferencias de tiempo entre O1 y O2 son aparentemente inexistentes, pudiendo ser resultado de que no se han podido aplicar las técnicas de optimización añadidas por O2 debido a las características de código.

Por último, tenemos a O3, el cual mejora en gran medida el tiempo de ejecución, pudiendo concluir que las técnicas de optimización que implementa son altamente eficaces en este código concreto.

Código ensamblador

Traducción del compilador

A continuación compararemos los códigos ensambladores obtenidos de la compilación con la opción -S de gcc. También se han traducido los programas a instrucciones de MIPS64 utilizando la página Compiler Explorer, pero estas traducciones no se referencian en la tabla y solo se usan para mejorar el entendimiento del código.

En la siguiente tabla se muestra la opción usada, las líneas de código y el número de bloques básicos utilizados obtenidos de la traducción mediante el compilador.

OPCIÓN DE COMPILACIÓN	LÍNEAS DE CÓDIGO	NÚMERO DE BLOQUES BÁSICOS
-O0	336	38
-O1	285	35
-O2	286	32
-O3	328	34
-Os	259	32

Se observa que el programa no optimizado tiene el mayor número de bloques básicos, pero no se presenta mucha diferencia respecto del resto de programas. También reducen el tamaño del programa, reduciendo el número de líneas.

También observamos como se reducen el número de bloques básicos conforme avanzamos en las opciones, implicando una mejora de rendimiento asociada a una mayor libertad de reorganización. No obstante, la opción O3 no reduce el número de bloques respecto a O2. Este hecho puede venir derivado de las optimizaciones adicionales utilizadas, las cuales sacrificarían el número de bloques a cambio de un resultado con mayor eficiencia temporal.

MIPS64 según Compiler Explorer

Los códigos de instrucciones del MIPS64 muestran unas claras diferencias en la estructuración de las instrucciones. El código no optimizado presenta las instrucciones secuencialmente según la aparición de las líneas de código asociadas. No obstante, las diferentes opciones de compilación realizan optimizaciones que reestructuran el código y juntan bloques básicos, mejorando potencialmente el rendimiento de los programas. Mientras que O1 y Os realizan reestructuraciones cautelosas, sin separar demasiado o nada las instrucciones referentes a una misma línea de código, las optimizaciones de O2 y O3 mueven las instrucciones independientemente de la línea asociada y observando que instrucciones asociadas llegan a estar separadas por una gran cantidad de líneas de código.

Discrepancias entre traducciones

Las traducciones observadas en el MIPS64 no se corresponden fielmente con las traducciones generadas por el compilador, pues en el MIPS64 las optimizaciones de O2 y O3 no presentan diferencia alguna, mientras que la traducción del compilador nos dice lo contrario. Esto puede deberse a las discrepancias entre nuestro compilador y el presente en la página utilizada, más las bases del código siguen siendo las mismas.

Conclusiones

Se ha observado que el código no optimizado tiene un rendimiento pobre respecto de los códigos optimizados.

Las opciones de compilación O1, O2 y Os tienen un rendimiento similar y sus códigos ensamblador asociados son bastante similares. No obstante, la opción O3 no reduce demasiado el tamaño del archivo, pero debido a este sacrificio de tamaño se mejora enormemente el rendimiento.

En resumen, el código compilado con las opciones de optimización mejoran en gran medida el rendimiento del programa. No obstante, las opciones de optimización que añade O2 no ayudan a mejorar el programa, teniendo un rendimiento incluso un poco peor que las opciones O1 y Os. Por último, la opción O3 añade opciones de compilación que, aunque sacrifiquen tamaño del ejecutable, mejoran enormemente el rendimiento del programa respecto del resto de opciones de optimización.

Compilación con -funroll-loops

En este apartado compilaremos un código proporcionado un total de dos veces con la opción -O1. En la segunda compilación añadiremos la opción de optimización -funroll-loops. Realizaremos una serie de mediciones y comparativas relativas a esta opción.

```c
#include <stdio.h>
#define N 10000
double res[N];
main() {
    int i;
    double x;
    for (i = 0; i < N; i++)
        res[i] = 0.0004 * i;
    for (i = 0; i < N; i++) {
        x = res[i];
        if (x < 10.0e6)
            x = x * x + 0.0004;
        else
            x = x - 900;
        res[i] += x;
    }
    printf("resultado= %e\n", res[N - 1]);
}
```

Código 2 - Código base para medir el rendimiento de la opción -funroll-loops

El array de resultados final ha sido creado utilizando memoria dinámica para poder utilizar un rango de valores de N mayor.

Las características de la máquina empleada para realizar las pruebas se pueden consultar en el apartado Anexo - Máquina de pruebas.

Tiempos de ejecución variando N

Hemos realizado una serie de mediciones donde podremos comparar el rendimiento de la opción de optimización -funroll-loops variando los distintos tamaños de la N del programa. Se han realizado 20 pruebas con cada valor de N. Estos fueron 10000, 40000, 70000, 100000, 400000, 700000 y 1000000.

Los resultados obtenidos de las ejecuciones han sido tratados de antemano para eliminar los outliers encontrados, por lo que se considera que calcular el rendimiento medio es suficientemente representativo.

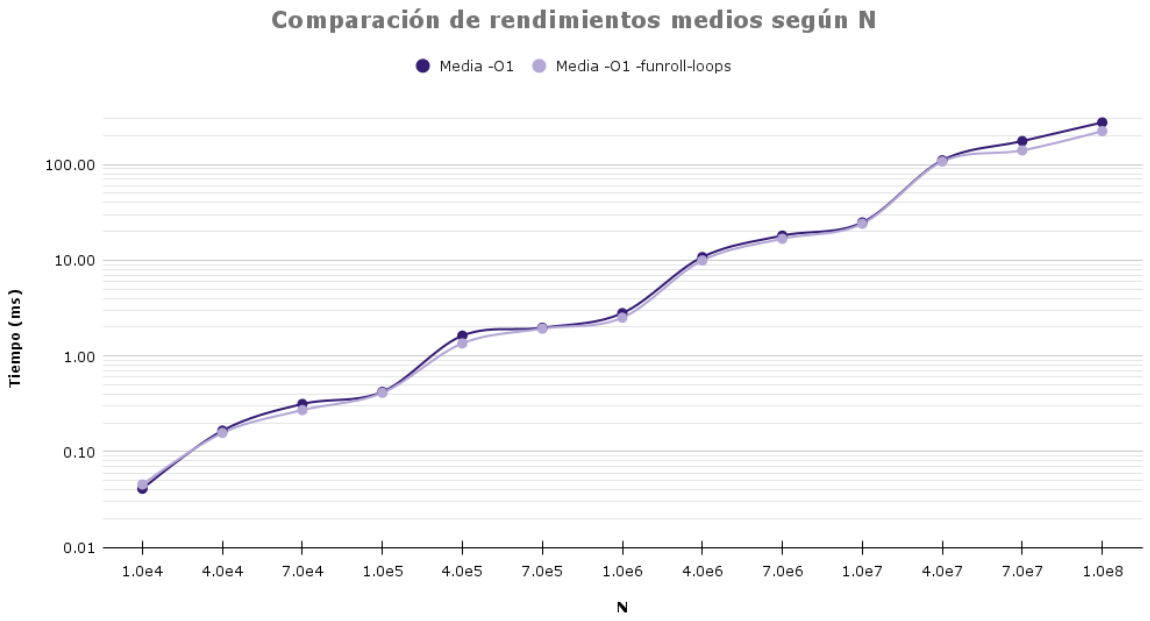

Figura 2 - Comparación de rendimientos

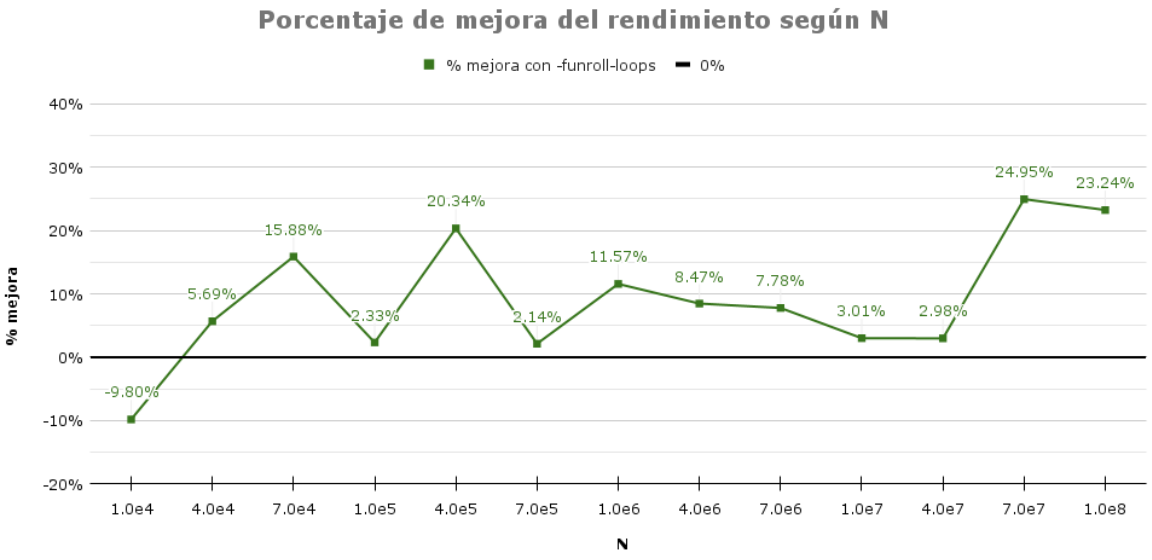

Figura 3 - Porcentaje de mejora respecto al código sin -funroll-loops

Como se puede observar en las dos gráficas, con el primer tamaño de N el código con -funroll-loops no mejora el rendimiento. El resto de tamaños presentan un crecimiento en el rendimiento de forma variable, teniendo un pico y continuando con un decrecimiento, repitiendo este patrón múltiples veces. Esto se puede observar mejor en la gráfica de *Porcentaje de mejora de rendimiento*.

Este crecimiento por picos puede tener relación con como realiza la optimización del código la opción de -funroll-loops según el tamaño de N, puesto que es una optimización dependiente del tamaño.

Código ensamblador

Traducción del compilador

A continuación compararemos los códigos ensambladores obtenidos de la compilación con la opción -S de gcc. También se han traducido los programas a instrucciones de MIPS64 utilizando la página Compiler Explorer, pero estas traducciones no se referencian en la tabla y solo se usan para mejorar el entendimiento del código.

OPCIÓN DE COMPILACIÓN	LÍNEAS DE CÓDIGO	NÚMERO DE BLOQUES BÁSICOS
-O1	253	33
-O1 -funroll-loops	445	54

En el código ensamblador se observa que la opción -funroll-loops añade una gran cantidad de código muy similar entre sí. Esto tiene que ver con el funcionamiento de esta opción de compilación, la cual "desenrolla" los bucles cuando su número de iteraciones es conocido en tiempo de ejecución. El desenrollar los bucles tiene como principal motivo el reducir el número de saltos que se realizan en el código y aumentar las operaciones que se realizan por iteración, mejorando el rendimiento.

Se observa que para la totalidad de lo valores de N utilizados -funroll-loops traduce el bucle inicial desenrollando 8 iteraciones del código base por cada iteración del código ensamblador.

En lo que respecta al desenrolle del segundo bucle, su traducción es más obtusa. Si se entra en el bloque if, se hace un desenrolle de 3 iteraciones, mientras que si no se hace, puede desenrollar hasta 4. Estos desenrolles se pueden combinar tal y como está dispuesta la traducción.

Conclusiones

Como se ha comentado en los anteriores apartados, el programa compilado con -funroll-loops mejora el rendimiento del programa, pero en una gran parte de valores de N, el rendimiento mejora en un porcentaje bastante bajo, siendo muy similar al rendimiento original.

La mejora que aporta -funroll-loops puede parecer buena a primera vista, mas hay que tener en cuenta que el desenrolle del segundo bucle implica la aparición de un mucho mayor número de saltos en cada iteración, afectando en gran medida al rendimiento del código. En este código ensamblador es más complicado para la CPU de realizar predicciones de salto, puesto que presenta múltiples saltos de forma continuada. No obstante, para el código sin la opción

de optimización es mucho más sencillo, pues solo presenta un salto en el interior del bucle y, debido a la naturaleza de la sentencia que contiene, es casi seguro que la predicción de salto realizada por el procesador será la correcta, recuperando en gran medida el tiempo perdido por el mayor número de iteraciones a llevar a cabo.

En conclusión, la mejora que proporciona -funroll-loops es notable en códigos que contengan un gran número de iteraciones en sus bucles, y se conozca en tiempo de compilación el número de iteraciones a realizar, pues su objetivo es desenrollar los bucles, aumentando el número de operaciones llevadas a cabo en cada iteración y reduciendo el número de estas. No obstante, tiene sus inconvenientes. Aunque tiene un gran potencial para mejorar el rendimiento de los códigos que cumplan con las características mencionadas, si los bucles están formados por una parte suficientemente grande de condicionales, el desenrolle de los bucles puede no resultar muy eficaz, llegando a empeorar el rendimiento. El desenrolle puede afectar a otros factores externos al código, como puede ser la predicción de saltos realizada por el procesador o incluso por tener que realizar accesos a memoria indeseados si el desenrolle no es lo suficientemente bueno. En definitiva, -funroll-loops ofrece una gran mejora siempre que se use en las situaciones adecuadas para ello.

Bibliografía

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

Anexo - Máquina de pruebas

Las pruebas de las diversas opciones de compilación 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	AMD Ryzen 5 2600 Six-Core Processor
Arquitectura	x86_64
Cache L1d	192 KiB
Cache L1i	384 KiB
Cache L2	3 MiB
Cache L3	8 MiB
RAM	16 GiB

Las pruebas sobre la opción de –funroll-loops 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