

# Compiladores e Intérpretes

## Práctica 1: Generación y optimización de código Optimizaciones del compilador en un producto de matrices

IGNACIO GARBAYO FERNÁNDEZ

Universidade de Santiago de Compostela

### Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Descripción de la técnica</b>	<b>2</b>
2.1. Configuración del experimento . . . . .	2
<b>3. Niveles de optimización de GCC</b>	<b>3</b>
<b>4. Programación y código en ensamblador</b>	<b>4</b>
4.1. Arquitectura de compilación y ejecución . . . . .	4
4.2. Bucles . . . . .	5
<b>5. Resultados obtenidos</b>	<b>6</b>
5.1. Tamaño de códigos objeto . . . . .	6
5.2. Tiempos de ejecución . . . . .	6
5.3. Aceleración . . . . .	6
<b>6. Conclusiones</b>	<b>6</b>

## 1. Introducción

El objetivo de este experimento es analizar el impacto de las distintas optimizaciones del compilador GCC sobre un código ineficiente de multiplicación de matrices cuadradas en lenguaje C. El código realiza el producto de dos matrices de tamaño  $n \times n$  utilizando tres bucles anidados sin ningún tipo de optimización algorítmica.

El programa se compila con los niveles de optimización -O0, -O1, -O2, -O3 y -Os, y se mide el tiempo de ejecución con precisión de nanosegundos. Además, se estudian las diferencias en el código ensamblador generado, el tamaño de los archivos de código objeto, los tiempos de ejecución y las aceleraciones obtenidas entre las versiones más eficientes.

Este análisis permite observar cómo las optimizaciones del compilador afectan al rendimiento y a la eficiencia del código sin modificar el algoritmo subyacente.

## 2. Descripción de la técnica

Para la práctica, se propone estudiar el rendimiento de la siguiente porción de código.

```
1  for(j=0;j<Nmax;j++) { /* Producto matricial */
2      for(i=0;i<Nmax;i++) {
3          t=0;
4          for (k=0;k<Nmax;k++) {
5              producto(A[i][k],B[k][j],&r);
6              t+=r;
7          }
8          C[i][j]=t;
9      }
10 }
```

### 2.1. Configuración del experimento

El objetivo del experimento es analizar el comportamiento de las diferentes optimizaciones del compilador, a saber: O0, O1, O2, O3 y Os. Para esto, se prueba con un pequeño *array* de valores de N. El *array* se muestra en (1).

$$N = (100, 300, 500, 600, 700.). \quad (1)$$

No se ha escalado en mayor medida el tamaño del problema, pues la definición estática en la pila de las matrices necesarias para la programación provocaba que se precisase más memoria de la disponible en la máquina. Además, definir las matrices como dinámicas empeoraba en gran medida la eficiencia de los accesos.

Al ya efectuarse el experimento mediante un doble bucle sobre cada uno de los valores de N mostrados en (1), no es necesario definir un número ITER para evitar el ruido, como sí se hace en la Práctica 2 de la asignatura.

Para cada valor de N, es fundamental ejecutar el algoritmo un número suficiente de veces, denotado por REPS, con el fin de obtener medidas de rendimiento representativas y estables. La variabilidad inherente al sistema —debido a factores como la planificación del sistema operativo, el estado de la caché, o procesos en segundo plano— puede provocar fluctuaciones significativas en los tiempos de ejecución individuales. Al repetir las mediciones y calcular posteriormente estadísticas como la **media**, la **desviación típica**, el **máximo** y el **mínimo**, se obtiene una caracterización más completa del comportamiento del algoritmo. Estas métricas permiten representar en las gráficas no solo la tendencia central,

sino también la dispersión y los extremos del rendimiento, ofreciendo una visión más realista y robusta para comparar implementaciones o analizar cuellos de botella. Para este particular, REPS se fija a 100, un número suficientemente grande, como se comentó en clase.

### 3. Niveles de optimización de GCC

El compilador GCC proporciona distintos niveles de optimización que pueden activarse mediante las opciones `-O0`, `-O1`, `-O2`, `-O3` y `-Os`, como se muestra en [1]. Cada uno de estos niveles activa diferentes transformaciones y análisis sobre el código fuente con el objetivo de mejorar su rendimiento o reducir su tamaño, a costa de aumentar el tiempo de compilación o dificultar la depuración.

El nivel `-O0` es el predeterminado y está diseñado para acelerar la compilación y facilitar la depuración. No se realiza ninguna optimización sobre el código: se generan instrucciones prácticamente una a una a partir del código fuente. Esto permite una correlación directa entre el código fuente y el ejecutable, pero da lugar a un binario ineficiente y con tiempos de ejecución mayores.

El nivel `-O1` introduce un conjunto básico de optimizaciones que buscan reducir tanto el tamaño como el tiempo de ejecución del código sin aumentar demasiado el tiempo de compilación. Entre las optimizaciones activadas en este nivel se incluyen:

- Eliminación de código muerto, es decir, instrucciones que nunca se ejecutan o cuyos resultados no se utilizan.
- Propagación de constantes, que sustituye expresiones por sus valores constantes cuando sea posible.
- Eliminación de operaciones redundantes, como comparaciones duplicadas o asignaciones innecesarias.
- Fusión de ajustes de pila consecutivos para mejorar la eficiencia en llamadas a funciones.
- Omisión del puntero de marco en funciones simples, lo que reduce el uso de registros.

Este nivel es especialmente adecuado para programas grandes generados automáticamente, donde se desea un equilibrio entre rendimiento y consumo de memoria durante la compilación.

El nivel `-O2` amplía las optimizaciones anteriores y activa casi todas las que no suponen un compromiso directo entre tamaño y velocidad. Se añaden optimizaciones más complejas, como:

- Inlining parcial de funciones pequeñas o llamadas únicas, lo que evita la sobrecarga de llamadas.
- Hoisting de cargas adyacentes, que agrupa lecturas similares para reducir accesos a memoria.
- Reordenamiento de bloques de código y funciones para mejorar la localidad de instrucciones.
- Vectorización de bucles, que transforma operaciones escalares en operaciones vectoriales cuando es posible.
- Eliminación de punteros nulos redundantes y optimizaciones para cadenas de texto y cálculos de longitud.

`-O2` es probablemente el nivel más equilibrado en la mayoría de los casos reales, ya que mejora sensiblemente el rendimiento sin aumentar demasiado el tamaño del ejecutable.

El nivel `-O3` activa todas las optimizaciones de `-O2` y añade otras aún más agresivas orientadas a maximizar la velocidad de ejecución, aunque esto puede aumentar considerablemente el tamaño del código generado. Entre las optimizaciones adicionales destacan:

- Desenrollado y reordenamiento de bucles, lo que permite aprovechar mejor la jerarquía de memoria.
- Separación y duplicación de caminos condicionales para mejorar la predictibilidad de las ramas.
- Vectorización avanzada con un modelo de costes dinámico para adaptar las transformaciones al hardware.
- Clonado de funciones para especializar comportamientos en contextos diferentes.

-O3 puede producir importantes mejoras de rendimiento en código intensivo en cálculos, aunque no siempre es beneficioso para todos los programas.

Finalmente, el nivel -Os está diseñado para minimizar el tamaño del ejecutable. Incluye todas las optimizaciones de -O2, excepto aquellas que tienden a aumentar el tamaño del código, como el alineamiento agresivo de funciones o bucles. Además:

- Se fuerza la inserción de funciones en línea solo si esto reduce el tamaño total del código.
- Se reorganiza el código para maximizar la reutilización de instrucciones y estructuras de control.
- Se ajusta el uso de memoria intermedia para reducir la huella del programa.

-Os es ideal para sistemas embebidos o programas donde el tamaño del binario es crítico.

Cada nivel representa una estrategia diferente en cuanto a las prioridades del programador: depuración (-O0), compilación rápida (-O1), rendimiento general (-O2), rendimiento máximo (-O3) o tamaño mínimo (-Os).

## 4. Programación y código en ensamblador

### 4.1. Arquitectura de compilación y ejecución

La compilación con la opción -S de GCC produce códigos fuente del ensamblador de la máquina mediante los que podemos analizar la traducción de las 2 estrategias. Se han ejecutado los códigos en una máquina ASUS Zenbook UX425EA 1.0, equipada con el sistema operativo Ubuntu 22.04.3 LTS x86\_64. Cuenta con un procesador Intel i7-1165G7 de 4 núcleos físicos de undécima generación, que funciona a una frecuencia de reloj de 2.80 GHz. Las direcciones de memoria ocupan 39 bits físicos y 48 bits virtuales.

Cada núcleo dispone de 2 hilos de procesamiento, resultando en un total de 8 hilos. Dispone de 16 GiB de memoria RAM totales y de 4 niveles de memoria caché: L1d (192 KiB, 4 instancias), L1i (128 KiB, 4 instancias), L2 (5 MiB, 4 instancias) y L3 (12 MiB, 1 instancia). El TDP (*Thermal Design Power*) es configurable entre 12 y 28 W. Se omite la inclusión de otros componentes como la tarjeta gráfica o de sonido porque no afectarán a los resultados de las pruebas. La versión del compilador utilizada es GCC 11.4.0.

Con el fin de ejecutar las dos versiones en las mismas condiciones, el envoltorio de declaraciones y variables de medición previos a las llamadas correspondientes ha sido el que se muestra a continuación.

Conviene mencionar que las mediciones se han efectuado utilizando la función `clock_gettime()`, pues aporta una precisión de nanosegundos, necesaria para poder comparar las versiones compiladas con O2, O3 o Os.

```
1 float A[Nmax][Nmax], B[Nmax][Nmax], C[Nmax][Nmax], t, r;
2 int i,j,k;
```

```

3
4 // Abrir fichero de escritura
5 if (argc < 2) { ... }
6 nombre_archivo = argv[1];
7 FILE* file = fopen(nombre_archivo, "a");
8 if (file == NULL) { ... }
9
10 // Calentamiento de la caché
11 for(i=0;i<Nmax;i++) { /* Valores de las matrices */
12     for(j=0;j<Nmax;j++) {
13         A[i][j]=(i+j)/(j+1.1);
14         B[i][j]=(i-j)/(j+2.1);
15     }
16 }
17
18 // Medir el overhead
19 clock_gettime(CLOCK_MONOTONIC, &overhead);
20 clock_gettime(CLOCK_MONOTONIC, &inicio);
21 // algoritmo
22 clock_gettime(CLOCK_MONOTONIC, &final);
23
24 // Cálculo de tiempos
25 tiempo_overhead = (inicio.tv_sec - overhead.tv_sec) + (inicio.tv_nsec - overhead.tv_nsec) / 1e9;
26 tiempo = (final.tv_sec - inicio.tv_sec) + (final.tv_nsec - inicio.tv_nsec) / 1e9;
27
28 // Imprimir en el archivo
29 fprintf(file, "%d\t%.9f\n", Nmax, tiempo);
30 // Cerrar el archivo
31 fclose(file);
32 return 0;

```

## 4.2. Bucles

Vamos a centrarnos en la traducción de los bucles de algoritmo. La versión con 00 presenta la estructura siguiente.

1

Por otra parte, con 01 destaca lo que se expone a continuación.

1

Para 02, es relevante el siguiente bloque de código.

1

La compilación con 03 introduce los cambios que se comentan de seguido.

1

Finalmente, 0s produce los cambios que se muestran a continuación.

1

## 5. Resultados obtenidos

Los experimentos se han realizado teniendo en cuenta los parámetros  $N$  y REPS explicados en la sección 2. Las fases del experimento también se han introducido en la sección 4. En primer lugar, se realiza el calentamiento de la caché. Las matrices se llenan de valores flotantes (no enteros) para que ya puedan estar disponibles en niveles altos de la caché.

A continuación, se obtiene el *overhead*, medición que tiene como objetivo estimar el tiempo que tarda en ejecutarse la propia medición del tiempo. Esto se aproxima calculando el tiempo que se tarda en usar `clock_gettime()` dos veces seguidas.

### 5.1. Tamaño de códigos objeto

### 5.2. Tiempos de ejecución

### 5.3. Aceleración

Una mejor manera de comparar las dos versiones del código es obtener la ganancia de una versión frente a otra, es decir, la aceleración, que vendrá dada por:

$$ac = \frac{t_{\text{una versión}}}{t_{\text{otra versión}}}. \quad (2)$$

Para cada tamaño de  $N$ , se consideran los tiempos de ejecución correspondientes a dos versiones distintas de un programa. Sea  $T_{\text{OPT}}^{(N)}$  y  $T_{\text{SIN}}^{(N)}$  los conjuntos de tiempos para cada versión. Se definen los cuantiles como  $Q_p(X)$ , siendo  $Q_{0.25}(X)$  el cuantil inferior (primer cuartil) y  $Q_{0.75}(X)$  el cuantil superior (tercer cuartil) del conjunto  $X$ . A partir de estos cuantiles, se estiman límites robustos para el *speedup* como una franja entre dos cocientes: el inferior, que representa una ganancia conservadora, se calcula como:

$$\text{Speedup}_{\min} = \frac{Q_{0.25}(T_{\text{SIN}}^{(N)})}{Q_{0.75}(T_{\text{OPT}}^{(N)})}, \quad (3)$$

y el superior, que representa una ganancia potencial más optimista, como

$$\text{Speedup}_{\max} = \frac{Q_{0.75}(T_{\text{SIN}}^{(N)})}{Q_{0.25}(T_{\text{OPT}}^{(N)})}. \quad (4)$$

Esta franja proporciona una estimación del rendimiento relativa entre ambas versiones, mitigando la influencia de valores atípicos.

## 6. Conclusiones

## Referencias

- [1] Free Software Foundation, Inc., “GCC Optimize Options,” *A GNU Manual*, 1988-2025. dirección: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-00>.