Generación y Optimización de Código

En este documento exploraremos algunas de las opciones que nos dan compiladores como Clang o GCC a la hora de obtener un código ejecutable que busque optimizar algún parámetro en concreto, sea este velocidad de ejecución o de compilación, tamaño del ejecutable o integridad y seguridad del programa entre otros. Comenzaremos por obtener las diferentes etapas por las que pasa un código en alto nivel hasta llegar a ser ejecutable para luego explorar un poco más en detalle la técnica de desenrollo de bucles que aplica GCC.

1.- Opciones de Generación de Código en Clang

Pese a que en el documento se define el ejercicio sobre el compilador GCC nosotros hemos realizado este apartado con el compilador por defecto en MAC OS, Clang. La razón de esto es que, además de no tener GCC disponible a la hora de realizar el ejercicio, al profesor le pareció interesante explorar que otras características nos podría aportar un compilador pensado para los mismos fines que GCC pero internamente distinto y aprender sobre la variedad de opciones a escoger en la actualidad.

El ensamblador generado por Clang es diferente que el IA32 que genera GCC, clang genera un código intermedio de bajo nivel conocido como LLVM IR. Esto se debe a que Clang está construido por encima de la plataforma LLVM, una colección de módulos y librerías enfocadas a la generación y optimización de código de forma universal, de forma que son capaces de generar código para prácticamente cualquier hardware actual y prácticamente todo el hardware antiguo. Esto lo consiguen mediante el uso de la representación intermedia LLVM IR que luego usan para generar código ejecutable específico para la plataforma solicitada. Sobre LLVM se han construido varios compiladores de uso extendido como dragonegg (Ada y Fortran) ya que automatiza la etapa tanto de optimización de código como la de generación del mismo para una máquina concreta.

1.1.- Código utilizado

Para este apartado utilizaremos un código de que inicializa dos matrices a unos valores dados para luego multiplicarlas entre sí, el código completo se puede ver en el repositorio asociado a este apartado[1]. En dicho código se presentan opciones para imprimir las matrices y el resultado así como para contar el tiempo de ejecución usando la función “gettimeofday”.

1.2.- Opciones del ensamblador

Pese a que no se pide que se expliquen los apartados en los que se prueban las diferentes opciones de ensamblador (-E, -S, -c y -static) comentaremos que en esta sección hemos visto diferencias entre GCC y Clang, sobre todo en el código ensamblador que podremos ver en el repositorio[2]. Las diferencias se deben al uso de LLVM IR en lugar de IA32.

1.3.- Opciones de Optimización

En esta sección analizaremos los resultados obtenidos de las ejecuciones del cidigo con diferentes niveles de optimización que nos permite el compilador. Para ello comentaremos punto por punto cada una de ellas. En las notas de la página pondremos un enlace a la tabla de resultados obtenida[3].

- **-O0**: Esta opción es la que nos da peores resultados en términos de tiempo y tamaño del código objeto. Esto se debe a que especificando esta opción el compilador no realizará absolutamente ninguna mejora sobre nuestro código, simplemente traducirá lo que tenemos a un ejecutable final. Útil para operaciones de debug pero no tanto si buscamos un buen rendimiento.

---

[1] Se puede ver el programa en: /src/testing_MatrixProduct.c

[2] Se pueden ver todos los ensambladores generados para las matrices en: /bin/ens

[3] Tabla de resultados: https://docs.google.com/spreadsheets/d/1XdohO1pggRYntl6y-39ZpRUc-Buel13R-v5UEB1DL2Mk/pubhtml

• **-O1**: En este nivel de optimización llegamos a disminuir el tamaño del código objeto en unos 300 bytes (de 2000B a 1700B) además de mejorar el tiempo de ejecución hasta casi reducirlo hasta la mitad. Esto es debido a que en esta opción se aplican varias técnicas de optimización que no requieren decidir entre tamaño del ejecutable y velocidad. Aún así no se aplican las técnicas de optimización más costosas como la reprogramación de instrucciones (instruction scheduling).

• **-O2**: Esta es la opción por defecto usada en sistema GNU. Con este nivel se aplican las técnicas de -O1 sumadas a algunas otras consideradas "más costosas" como la reprogramación de instrucciones. Las optimizaciones aplicadas no incrementan el tamaño de ejecutable en favor de la velocidad por lo que esta opción nos da el mínimo tamaño de programa con la máxima optimización. Como podemos ver en la tabla, diminuimos el tiempo de ejecución en más de la mitad desde -O1 y reducimos el tamaño del ejecutable en unos 500 bytes (hasta 1200B).

• **-O3**: Este es el nivel de optimización más alto soportado por GCC (y Clang) en el que se aplican todas las técnicas posibles en favor de la velocidad de ejecución del programa aunque esto implique aumentar en tamaño del ejecutable. Pese a esto es posible que cuando estas técnicas no son ideales para el tipo de programa a compilar se obtenga como resultado un código más lento. Por estas razones y algunas otras como problemas de concurrencia y demás, se recomienda siempre probar la velocidad y tamaño de los programas con -O3 y -O2 para quedarnos con la mejor opción. En nuestro caso hemos obtenido un tamaño de ejecutable y tiempos de ejecución virtualmente idénticos causados seguramente por la simplicidad del código y el hecho de que ninguna de las técnicas de -O3 aplica en este caso.

• **-Os**: Esta opción de compilación busca el menor tamaño de ejecutable posible para lo cual utiliza diversas técnicas de optimización. El objetivo es obtener programas pensados para sistemas con pocos recursos de memoria. En algunos casos se llega a obtener código más rápido que con las otras opciones debido al mejor uso de la caché. Este nuestro caso da la sensación de que no se han aplicado técnicas que hayan afectado mucho al ejecutable pues obtenemos tiempos y tamaños casi iguales a -O3 y -O2. Si se ve una reducción de tamaño en el caso de que compilemos el código que muestra las matrices.

2.- Desenrrollo de bucles (“Loop-Unrolling”)

Este apartado no hemos podido realizarlo en Clang pues no comparte las mismas opciones de desenrrollo de bucles que GCC y no conseguíamos obtener resultados relevantes. En GCC hemos desarrollado un código muy simple que contiene dos bucles con operaciones matemáticas sencillas dentro que podremos ver en el repositorio[4]. Además también podremos ver la tabla de tiempos obtenidos en el enlace a la tabla previamente citada[5]. Finalmente, se podrán ver los código ensamblador a los que nos referiremos en el repositorio[6].

1.1.- Análisis del Código Ensamblador

La diferencia más notable entre los dos programas es que, como uno se podría esperar, cuando especificamos la opción de “loop-unrolling” obtenemos unos tamaños tanto de ejecutable como de archivo del código en ensamblador mucho más grandes y esto es debido a que vemos como entre la etiqueta del bucle hasta el salto que vuelve a esa etiqueta hay mucho más código que en el programa sin “unrolling”. Ese código simplemente son repeticiones de las mismas operaciones una y otra vez que se hubieran hecho en diferentes iteraciones del bucle pero que al desenrollarlo realizamos en la misma iteración. Como dato interesante podríamos comentar que al observar los ensambladores de programas con pocas iteraciones (2, 5, 10) podemos ver que ni siquiera existe bucle alguno sino que se ponen todas las “iteraciones” una detrás de otra sin ningún tipo de salto. En los programas con más de iteraciones vemos como si tenemos etiquetas para el bucle y una instrucción de salto pero entre ambas tenemos varias iteraciones “desenrolladas”, es decir, todas secuencia les dentro de una iteración real del bucle. Como es lógico, en los ensambladores en los que no hacemos “loop-unrolling” no vemos ninguna diferencia, solamente en valor con el que se compara para terminar el bucle (entre diferentes tamaños de N) ya que en todas las iteraciones del bucle del ensamblador se hace una iteración del bucle de alto nivel.

---

[4] Código para probar los bucles: /src/testing_unrollLoops.c

[5] Tabla de resultados: https://docs.google.com/spreadsheets/d/1XdohO1pggRYntl6y-39ZpRUc-Buel13R-v5UEB1DL2Mk/pubhtml

[6] Directorio de los códigos en ensamblador: /bin/ens

Otra nota interesante es que parece que la técnica busca un número de iteraciones antes de un salto que se adecue al tamaño del problema de una forma inteligente. La prueba es que para N=10 tenemos 10 iteraciones antes de un salto pero para N=1000 y N=10000000 tenemos 8 iteraciones antes de un salto.

1.2.- Análisis del Tiempo de Ejecución

En la tabla antes citada podemos ver como los tiempos de ejecución varían entre los diferentes tamaños que hemos probado para N. Como era de esperar, para tamaños tan pequeños como 10 no vemos diferencia alguna, pero sin embargo cuando aumentamos el tamaño hasta 1000, 10000 y 10000000 si vemos una diferencia en velocidad apreciable.

Las variaciones de velocidad no parecen escalar de forma predecible y parece lógico atribuir las diferencias entre los programas con diferentes tamaños a simplemente la variación natural entre diferentes ejecuciones, no parecen estar causadas por ninguna razón relacionada con como la técnica trabaja con N iteraciones pues en los códigos ensamblador vemos que de desenrollan los bucles de la misma forma y con casi el mismo número de iteraciones antes de un salto.

Finalmente, la conclusión que obtenemos de estas mejoras es que a costa de tamaño del programa podemos realizar una sola instrucción de salto en lugar de varias, ahorrándonos el coste de dichas instrucciones. También parece fácil averiguar que las instrucciones de salto no parecen muy costosas en comparación con el contenido del bucle ya que aun desenrollando grandes bucles no obtenemos una mejora muy grande, sin embargo, alrededor de un 4% de mejor es un resultado que no se puede ignorar. Se podría recomendar esta técnica si el tamaño del programa no es un problema para el sistema y se realiza una gran cantidad de iteraciones sobre un bucle que contiene instrucciones poco costosas, pues el coste del salto es más importante en comparación.