-Nombre Oculto-
-Email Oculto-
28 de diciembre de 2016

Práctica 2: Generación Y Optimización De Código

Desenrolle de lazos internos y optimización de reducciones (Optimización 6)

En este documento se explicarán las utilidades de estas dos técnicas usadas conjuntamente para obtener mejor rendimiento a la hora de optimizar bucles. En los siguientes apartados veremos en que se basan estas técnicas para lograr mejores tiempos de ejecución para luego hacer una revisión de los resultados que hemos obtenido al aplicarlas a nuestro pequeño programa ejemplo. Finalmente comentaremos las conclusiones obtenidas durante nuestro proceso de análisis y daremos recomendaciones sobre cuando se deberían tener en cuenta este tipo de técnicas.

Tabla de contenido:

1.- Descripción de las técnicas utilizadas

Comenzaremos con una breve explicación en la que veremos en que consisten cada una de las técnicas que se van a aplicar. Primero nos centraremos en el desenrolle de lazos para luego comentar como aplica la optimización de reducciones al combinarlas convenientemente.

1.1.- Desenrolle de lazos internos

Esta familiar técnicas con la que hemos trabajado en prácticas anteriores es un ejemplo de utilización de técnicas de intercambio espacio-tiempo pues elegimos aumentar el tamaño de nuestro código ejecutable a cambio de aumentar la velocidad del mismo a la hora de ser ejecutado. Esto se consigue copiando el contenido de un bucle presente en el código original sin optimizar y repitiéndolo varias veces de forma secuencias de forma que no se tengan que ejecutar las instrucciones relacionadas con el bucle al final de cada una de esas secuencias de código. De esta forma hacemos que las instrucciones asociadas al bucle (comprobaciones de fin de bucle, incremento de contadores, instrucciones de salto, etc.) se realicen una fracción de las veces que se hubieran realizado sin optimizar.

Si o bien el programador a la hora de hacer el código o bien el compilador a la hora de optimizar, son capaces de precalcular cuantas iteraciones se van a realizar o si el número de iteraciones puede ser dividido de forma exacta por un factor concreto, se puede aplicar esta técnica para reducir el overhead asociado a las instrucciones de bucle explicadas previamente.

Al utilizar esta técnica con éxito obtenemos muchos potenciales beneficios:

- Las reducciones en tiempo de ejecución al ahorrarnos instrucciones del bucle deberían compensar las desventajas de tener un ejecutable más grande.
- Mejoras en las predicciones de salto por parte del procesador pues se puede reducir el numero de repeticiones para que no sobrepase el número máximo de predicciones permitida por el procesador.
- Vectorización: Se puede hacer cuadrar el número de iteraciones con el tamaño de vector a usar para hacer uso de los beneficios de la vectorización.
- Si el bucle tiene un número de iteraciones lo suficientemente bajo se podría eliminar completamente la existencia de instrucciones de bucle desenrollando por completo y eliminando así el overhead asociado al mismo.

Sin embargo podemos encontrarnos con que no obtenemos los beneficios deseados al aplicar la técnica, esto se puede deber a alguna de las siguientes desventajas que presenta:

- El código ocupa más espacio en memoria por lo que es posible que se produzcan más fallos de caché al intentar leer nuestro programa que tengan más peso que las mejoras obtenidas por la técnica, haciéndola contraproducente. Para detectar esto no se puede probar el bucle aislado sino que se tiene que tener en cuenta el resto del programa y el entorno en el que deberá funcionar finalmente.
- Algunos procesadores tienen un buffer para aumentar la velocidad de bucles de pequeño tamaño en el que guardan las instrucciones que se encuentran dentro del mismo. Este buffer suele tener un tamaño de unas 20 a 40 instrucciones y es posible que al aplicar desenrolle de bucles hagamos que estos superen ese límite de iteraciones, haciendo este buffer inservible y diminuyendo el rendimiento.
- Si el número de iteraciones del bucle no es divisible por un factor deseable a la hora de desenrollarlo podemos encontrarnos con códigos muy poco legibles, muy complicados y con muchos saltos. Esto ocurre por que las iteraciones sobrantes luego de desenrollar con un factor determinado (numero de iteraciones del antiguo bucle dentro del nuevo bucle) tendremos que introducir por otro lado (ya sea antes o después del bucle desenrollado) las instrucciones que no se han realizado en el bucle.
- El bucle desenrollado puede necesitar una cantidad de registros superior.

1.2.- Optimización de reducciones

Esta técnica es ampliamente conocida como “Rotura de cadenas de dependencia”. Consiste en aprovecharse de las capacidades de los procesadores para ejecutar instrucciones fuera de orden y en paralelo en el “pipeline” del procesador al romper una cadena de dependencia existente en nuestro código. Una cadena de dependencias se crea cuando en varias instrucciones secuenciales se necesita un resultado de la instrucción anterior para ejecutar la siguiente, haciendo que dichas instrucciones tengan que ser llevadas a cabo en un orden estricto.

Al utilizar esta técnica se rompen las dependencias presentes entre estas operaciones para permitir que las instrucciones sean ejecutadas en el orden que más le convenga al procesador, frecuentemente aumentando el rendimiento debido a una mejor planificación por parte del “scheduler” o planificador del procesador.

Esta técnica no es tan fácil de reconocer visualmente como otras presentes en esta práctica pues sus efectos pueden conseguirse mediante construcciones de código extremadamente diferentes siempre que pretendan romper cadenas de dependencia presentes en él. Por

ejemplo, en el código usado en nuestra experimentación estamos separando una instrucción original de un bucle que contiene una suma y una multiplicación en cuatro instrucciones que recogen el resultado en cuatro variables distintas que luego se juntan de la siguiente forma:

Se pasa de el siguiente fragmento (cuerpo del bucle):

```
a = a + x[i] * y[i];
```

A esto:

```
a = a + x[i] * y[i];
a1 = a1 + x[i+1] * y[i+1];
a2 = a2 + x[i+2] * y[i+2];
a3 = a3 + x[i+3] * y[i+3];
```

Juntando los resultados al terminar el bucle con:

```
a = a + a1 + a2 + a3;
```

De esta forma estamos rompiendo la cadena de dependencias original en la que cada iteración del bucle necesitaba tener el valor anterior de a para calcular el siguiente. En el nuevo bucle se pueden calcular fuera de orden las instrucciones mostradas, además puede haber dentro del pipeline varias de estas instrucciones a la vez pues unas no dependen del resultado de otras de ninguna forma.

Sin embargo esta no es la única forma de romper cadenas de dependencia, de hecho podríamos romper otra cadena de dependencia en este mismo código simplemente cambiando la última instrucción mostrada por:

```
a = (a + a1) + (a2 + a3);
```

Ya que si suponemos que el procesador está sumando primero a y a1, luego a ese resultado le suma a2 y luego a ese a3 podemos eliminar la dependencia que hace que tengamos que hacer las sumas de esa forma introduciendo paréntesis de forma que se puedan hacer las dos sumas fuera de orden y en paralelo y luego se sumen los dos resultados.

Realmente, romper cadenas de dependencia no tiene ninguna desventaja per se ya que depende de como se consiguen romper dichas cadenas. En este caso las desventajas son el uso de más variables y de una serie de instrucciones más que hacen la suma de los resultados parciales obtenidos.

Nota: Esta última optimización mencionada no está presente en el código final pues en el enunciado de la práctica se especifica cual es el código original y optimizado.

2.- Resultados obtenidos

Al aplicar los scripts referenciados en el apartado de recursos utilizados hemos obtenido una serie de datos que hemos introducido en una hoja de cálculo en la plataforma online Drive. A la hora de obtener datos hemos seguido dos aproximaciones diferentes, primero hemos utilizado una serie de valores balanceados para el número de iteraciones que realizaremos del problema y para el tamaño del problema N. De esta forma, con un N pequeño hemos aumentado el número de iteraciones para hacer que el programa tardara un tiempo relevante en terminar eliminando así posibles errores de redondeo en las medidas. Sin embargo, con esta aproximación no tiene sentido comparar el tiempo entre distintos valores de N pues no es la única variable que cambia. Para solucionar esto hemos realizado otra serie de pruebas en las que siempre realizamos el mismo número de iteraciones y vamos variando el valor de N. En los datos que analizaremos nos centraremos en los números asociados al ejecutable con las técnicas introducidas en nuestro código ya que no hemos encontrado una opción de compilador que englobara las dos técnicas usadas en este ejercicio. Pese a que no había una técnica del compilador, hemos introducido los datos obtenido mediante el uso de "-funroll-loops".

Tanto la gráfica de datos balanceados[1] como la tabla de datos con las mismas iteraciones[2] se encontrarán en un enlace al pié de esta página y en el apartado de recursos utilizados.

En los siguientes apartados analizaremos las diferencias que vemos según variemos el tamaño del problema y luego examinaremos las diferencias con los diferentes niveles de optimización que nos aporta el compilador.

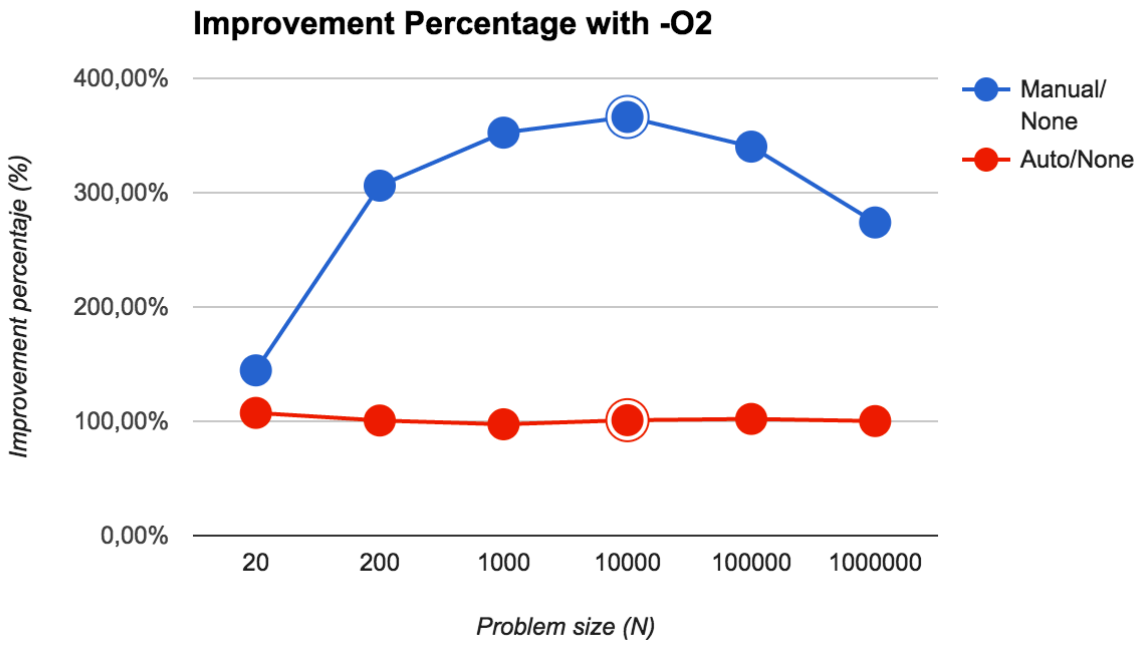
2.1.- Variando el tamaño del problema

Utilizaremos como objeto de estudio principal la gráfica que muestra el porcentaje de mejora obtenido con la opción -O2 al utilizar las mejoras introducidas manualmente. Hemos optado por usar estos datos pues son los que muestran de forma más consistente las mejoras que nos aportan las técnicas de optimización y además es la opción que se usa normalmente por defecto a la hora de compilar con GCC. Nos centraremos en la línea que nos muestra la mejora del ejecutable generado con las técnicas introducidas de forma manual comparado con el original sin mejora alguna pues, como hemos dicho antes, la opción del compilador introducida (-funroll-loops) solo realiza desenrolle de bucles y no optimización de reducciones por lo que no tiene sentido comparar ambas.

---

[1] Enlace a la tabla de datos balanceados: https://docs.google.com/spreadsheets/d/1ebIpMh8sFr4a8NB3JFkt6uHXR9rvomwbf2ImGlTkSCI/pubhtml

[2] Enlace a la tabla de datos con las mismas iteraciones: https://docs.google.com/spreadsheets/d/1q-CLl_5cB_h9SNbkaTskK5miSR4eWh08fbZr9yPvJ7c/pubhtml



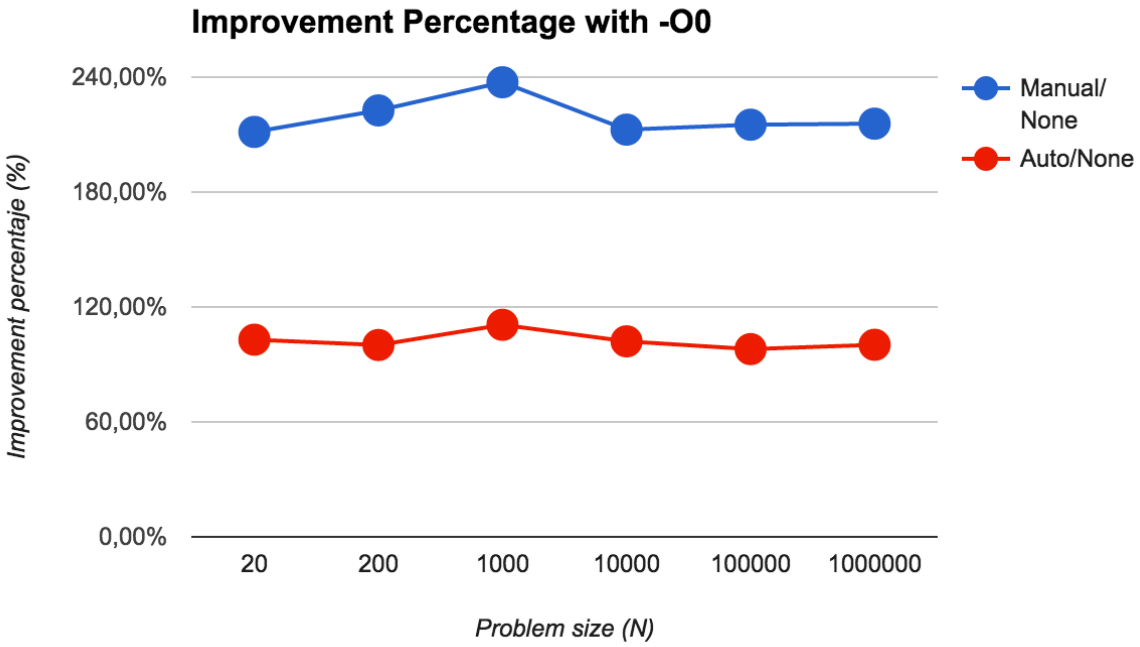
Como podemos ver en la línea azul, las técnicas aplicadas nos dan un porcentaje de mejora de alrededor al 300%, es decir, tres veces más rápido que la versión que no aplica ninguna técnica y, como indica la línea roja, también tres veces más rápido que la versión optimizada simplemente por el compilador.

Esto nos muestra que una de las dos técnicas está dando muy buenos resultados y es la de optimización de reducciones. Mediante nuestras propias pruebas hemos comprobado que si solo hacemos desenrolle de bucles no obtenemos mejoras realmente importantes comparado con las que estamos viendo en este caso, un ejemplo claro fueron los resultados de la práctica anterior en lo que veíamos como el desarrollo de bucles por parte del compilador si nos daba mejoras, pero no en la escala que vemos en este caso.

Primero analicemos la razón de una diferencia tan grande en general. La principal razón es el aprovechamiento del pipeline del procesador que nos permite paralizar varias instrucciones del bucle de forma efectiva pues estamos no solo desenrollando sino que estamos rompiendo las dependencias de las instrucciones dentro de este por lo que se pueden llevar a cabo fuera de orden y pueden estar todas dentro del pipeline, aumentando mucho la velocidad de ejecución del bucle como podemos ver.

La razón de que en tamaños muy pequeños no veamos una diferencia grande es que simplemente el coste del resto de código del programa es muy relevante comparado con el coste de las instrucciones del bucle pues hay muy pocas iteraciones del mismo. Al llegar a tamaños relativamente importantes (por encima de 200) ya vemos mejoras claras.

Algo que llama la atención es como en los tamaños muy grandes el porcentaje de mejora empieza a diminuir. Una de las razones es que cuantas más iteraciones y más larga es la duración del programa más probable es que suframos cambios de contexto entre diferentes procesos del sistema. Un cambio de contexto implica, entre otras cosas, guardar el estado de los registros actuales en memoria principal , esta operación resulta ser más costosa cuando trabajamos con más registros y en la versión mejorada de nuestro programa usamos un número de registros más elevado en que en la original. Otra es la forma en la que todas las optimizaciones asociadas a -O2 están interactuando con nuestro programa pues si echamos un vistazo a la siguiente gráfica que mostraremos a continuación (mejoras con -O0) vemos que los resultados son mucho más consistentes.



Aquí vemos como nuestras técnicas aplicadas por si solas tiene un efecto de mejora muy constante de un poco más del 200% (2 veces más rápido) incluso entre tamaños de N extremadamente diferentes. Al introducir otras mejoras del compilador mediante el uso de las opciones -O1, -O2 y -O3 dependemos de como dichas mejoras interaccionen con la versión mejorada y sin mejorar de nuestro programa y es posible que veamos variaciones difíciles de explicar como es el caso de los datos obtenidos.

2.2.- Diferentes opciones de compilación

Como podemos ver en las gráficas mencionadas anteriormente y enlazadas en el apartado 4, las optimizaciones que se introducen en cada uno de los niveles de optimización afecta de formas muy dispares a las diferentes versiones de nuestro programa, sean las versiones sin mejoras o la mejorada manualmente como la que introduce la opción del compilador que desenrolla bucles automáticamente.

La opción -O0 es la que más clara es a la hora de mostrarnos las mejoras introducidas manualmente, mostrándonos una mejora constante entre los diferentes valores de N y enseñándonos que la opción -funroll-loops no parece tener mucho efecto por si sola aplicada a la versión sin mejoras del programa. Esto es consistente con los resultados de la práctica 1, en la que se veían mejoras pero no muy relevantes.

Al aplicar -O1 empezamos a ver variaciones grandes en la línea del programa mejorado manualmente, en valores pequeños de N vemos mejoras importantes que se estabilizan con tamaños grandes de N. Por otra parte vemos como la opción -funroll-loops se complementa bien con las optimizaciones introducidas en -O1 dándonos una mejora de alrededor del 200%.

Con las opciones -O2 y -O3 vemos dos gráficas muy similares que indican que -funroll-loops no es relevante por si sola estando presentes las opciones de optimización más costosas de -O2 y -O3. Además la versión mejorada manualmente muestra los mejores porcentajes de mejora hasta ahora que parecen estar causados por alguna de las optimizaciones introducidas con estas dos opciones pero que parecen perder efecto al aumentar N, quedándose estancadas en tamaños muy grandes.

Llegar a determinar cuales de las optimizaciones introducidas por cada una de las opciones son las causantes de algunos fenómenos que vemos en las gráficas de -O1, -O2 y -O3 sería extremadamente complejo, más cuando las optimizaciones que se están aplicando pueden intelectual unas con otras y con las aplicadas por nosotros de forma manual de formas que no con perceptibles a simple vista incluso analizando el código ensamblador. Hemos observado que cuantas más optimizaciones se le permiten al procesador más difícil es entender lo que está ocurriendo en el código ejecutable final leyendo el código ensamblador asociado.

Es por esto que consideramos que la mejor opción para ver el potencial de las técnicas aplicadas es ver los datos asociados a la opción -O0 que aísla nuestras mejoras y las compara con una versión en la que no se han aplicado. Al ver su utilidad en casos herméticos y aislados podemos entender si nos son útiles en casos aislados y que pueden darnos mejoras aun mejores como podemos ver cuando dejamos que el compilador optimice por su cuenta.

3.- Conclusiones obtenidas y recomendaciones de uso

Al aplicar estas dos técnicas en conjunto sobre un bucle muy común se han observado resultados que a nivel personal me han parecido realmente sorprendentes. El aprovechamiento del pipeline del procesador y su capacidad para paralizar y realizar instrucciones fuera de orden es capaz de darnos mejoras realmente relevantes cercanas algunas veces a ser 3,5 veces mejores que el código original. Este tipo de números son realmente vistosos comparados con los que hemos obtenido cuando analizamos las técnica del desenrolle de bucles por si sola.

Sería interesante que, como ha sido mi caso, ayude a interesarse por las capacidades que tienen los procesadores actuales y a darnos cuenta de lo importante que es aprender a utilizar dichas capacidades. Al ver los resultados que se pueden obtener con técnicas de mejora que no conllevan demasiado coste en tiempo de programación ni tiene una extremada complejidad es posible que suframos una llamada de atención que nos haga preocuparnos por lo optimizado o poco optimizado que está nuestro código. Haciendo que realmente hagamos uso de nuestras capacidades como ingenieros para entender tanto la parte software como la parte hardware y como combinar ambas puede dar lugar a mejoras en eficiencia muy grandes.

En términos de recomendaciones de uso tenemos que hablar de forma independiente de desenrolle de bucles y de optimización de reducciones. Al leer los apartados de desventajas de desenrolle de bucles podemos ver claramente los casos en los que la técnica no se debe aplicar. Siempre que el tamaño del ejecutable pueda llegar a ser un problema debemos tener cuidado al aplicarla, además, si creemos que es posible que el ejecutable pase a ocupar más páginas en memoria causando fallos en caché también deberemos comprobarlo antes de usar la opción definitivamente. En general, si el tamaño del ejecutable puede ser un problema, se deberían de realizar pruebas en el entorno final de explotación.

Por otra parte, romper cadenas de dependencia será siempre beneficioso en sí mismo siempre y cuando se tenga en cuenta el coste de dicha rotura. Es decir, si se puede eliminar una dependencia sin introducir instrucciones nuevas entonces no habrá perjuicio alguno. Sin embargo es frecuente que, como en este caso, tengamos que introducir nuevas variables y nuevas lineas de código para romper las dependencias, si es coste de estas modificaciones es mayor que la posible ganancia entonces debemos descartarlas. Como de costumbre la mejor opción es hacer pruebas con ambas versiones si es posible, además de que cuanta más experiencias realizando dichas pruebas más se tenderá a hacer porciones de código con las mínimas dependencias desde un principio.

4.- Recursos Utilizados

- Código fuente adjunto a este documento:
    - En el directorio /scripts encontraremos los scripts para compilar (compile*) y ejecutar (run*) tanto la versión que varía las iteraciones como la que no (en cada uno de los directorios correspondientes).
    - En el directorio /src veremos el ejecutable usado para las experimentaciones: optimization_6.c.
    - El apartado /out tendrá un subdirectorio por opción de compilador que contendrá los ejecutables además de otro subdirectorio /logs con los datos obtenidos de las ejecuciones.
- Enlace a la tabla de datos balanceados: https://docs.google.com/spreadsheets/d/1ebIpMh8sFr4a8NB3JFkt6uHXR9rvomwbf2ImGlTkSCI/pubhtml
- Enlace a la tabla de datos con las mismas iteraciones (pulsar en la pestaña "Graphs" para ver las gráficas asociadas a los datos obtenidos): https://docs.google.com/spreadsheets/d/1q-CLl_5cB_h9SNbkaTskK5miSR4eWh08fbZr9yPvJ7c/pubhtml
- Optimizaciones en ensamblador: http://www.agner.org/optimize/optimizing_assembly.pdf
- Blog sobre desenrolle de bucles: http://blog.teamleadnet.com/2012/02/code-unwinding-performance-is-far-away.html