Compiladores e Intérpretes
Generación y Optimización de código

Índice

Especificaciones

El ejercicio se ha realizado en una máquina virtual con las siguientes características.

Ordenador

Las especificaciones en las que se han realizado las pruebas son las siguientes y se han obtenido con el comando *“lscpu”*:

- Arquitectura: x86_64
- modo(s) de operación de las CPUs: 32-bit, 64-bit
- Orden de los bytes: Little Endian
- CPU(s): 2
- Hilo(s) de procesamiento por núcleo: 1
- Nombre del modelo: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
- CPU MHz: 2394.456
- Caché L1d: 32K
- Caché L1i: 32K
- Caché L2: 256K
- Caché L3: 3072K

Sistema Operativo

El sistema operativo en el que se han realizado las pruebas es un Ubuntu versión 18.04.5 LTS

Compilador

La versión del compilador gcc empleado es la 7.5.0.

Optimización

La optimización a analizar es la de peeling de lazos, la 9ª optimización del pdf.

Características

El peeling de lazo o loop peeling es un tipo de optimización de código de los divisores de lazos o loops splitting.

Un loop splitting consiste en simplificar un bucle o eliminar dependencias rompiéndolo en múltiples bucles que tienen el mismo cuerpo pero iteran sobre diferentes porciones del rango de índices.

El loop peeling (1) es un caso especial de loop splitting que consiste en separar las iteraciones problemáticas del bucle y ejecutarlas fuera del bucle.

Para entenderlo mejor observemos nuestro código:

```
int i, k;
float  x[N], y[N];
for(k=0; k<ITER; k++)
   for(i=0; i<N; i++)
      if(i==N/2) x[i] = 0;
      else if(i==N-1) x[i] = N-1;
      else x[i] = x[i]+y[i];
```

Tenemos una declaración de variables y a continuación entramos en el bucle de ITER y dentro de este en el de N, este último es el que nos interesa. En este bucle recorremos N, de forma que hacemos un primer condicional en donde si i vale la mitad que N le asignamos a la posición i de x el valor 0. En caso de que i no valga N/2 comprobamos si i vale N-1, si es así le asignamos este valor a la posición i de x. Finalmente en caso de que nada de lo anterior sea cierto asignamos a la posición i de x la suma de x e y.

En este código lo que ocurre es que al recorrer el bucle ejecutamos para todas las iteraciones tres condicionales, comprobamos si i vale N/2, si vale N-1 y el caso de que nada de lo anterior sea, por lo que ralentizamos bastante la ejecución de cada iteración. Si optimizamos el código con loop peeling obtenemos el siguiente código:

```
for(k=0; k<ITER; k++)    {
   for(i=0; i<N/2; i++)
      x[i] = x[i]+y[i];
   x[N/2]=0;
   for(i=N/2+1; i<N-1; i++)
      x[i] = x[i]+y[i];
   x[N-1] = N-1;
}
```

Como podemos ver dentro del bucle de ITER ya no tenemos un único bucle que recorre N, sino 2 bucles. Lo que tenemos es un primer bucle que recorre hasta N/2, asignado a la posición i de x el valor de la suma de x e y; una vez llegamos a N/2 salimos del bucle y asignamos a la posición intermedia de x el valor de 0; volvemos a entrar en un bucle desde N/2 mas uno hasta justo la anterior posición de N, N-1, asignando a cada posición i de x la suma de x e y; una vez llegamos a N-1 salimos del bucle y asignamos a la última posición de x el valor N-1.

Como hemos visto en el ejemplo tenemos que asignar a la posición N/2 de x el valor 0 y a la última posición, N-1, de x el valor N-1, por lo que si ejecutamos un bucle hasta N tenemos

que ejecutar condicionales en todas las iteraciones; de forma que si ejecutamos dos bucles, uno hasta N/2 y otro desde N/2+1 hasta N-1, y asignamos fuera de los bucles los valores especiales no necesitamos realizar ningún condicional en los bucles, de forma que solo realizamos la asignación de suma de x.

Realizar este tipo de optimización supone la generación de múltiples bucles, por lo que tenemos que saber donde empezarlos y finalizarlos, lo que implica conocer a fondo el problema en cuestión. De esta forma los condicionales se encuentran fuera de los bucles lo cual agiliza los mismos, permitiendo que sean más rápidos.

GCC

Como información extra esta técnica de optimización fue introducida en el compilador de gcc en la versión 3.4.

Código fuente

Ambas implementaciones se han realizado en programas diferentes, compartiendo eso sí toda la estructura excepto el código a probar, tal y como se aprecia a continuación:

```
struct timeval inicio, final;
double tiempo, acumulado;

int main(){
        acumulado = 0.0;
        int N = 10;
        int ITER;
        int iteraciones;
        int repeticiones;
        printf("Que tamaño de ITER?");
        scanf("%d", &ITER);
        printf("Que cantidad de iteraciones por pruebas?");
        scanf("%d", &iteraciones);
        printf("Que cantidad de N desea probar?");
        scanf("%d", &repeticiones);

        for(int x = 0; x < repeticiones; x++){
                for(int y = 0; y < iteraciones; y++){
                        gettimeofday(&inicio,NULL);

                        //CODIGO
                        //FIN CODIGO

                        gettimeofday(&final,NULL);
                        tiempo = (final.tv_sec-inicio.tv_sec+(final.tv_usec-inicio.tv_usec)/1.e6);
                        acumulado += tiempo;
                }
                printf("N = %d, ITER = %d, iteraciones = %d, media tiempo = %f seg\n", N, ITER, iteraciones, acumulado/iteraciones);
                N = N * 10;
                acumulado = 0.0;
        }
}
```

En ambos programas se especifica un valor ITER constante, es decir, todas las iteraciones del bucle independientemente del valor de N tienen el mismo valor de ITER.

Ambos programas empiezan con 10 como valor de N y al finalizar cada bucle se multiplica N por 10, por lo que los valores a probar para N son múltiplos de 10. Los posibles valores para N entonces son diez, cien, mil, diez mil, cien mil, un millón, diez millones y así sucesivamente.

Al inicio de cada programa se pregunta: el tamaño de la variable ITER; la cantidad de iteraciones por prueba, esto es, cuantas veces se ejecuta el bucle para cada N y así poder

calcular la media para diferentes ejecuciones; y las veces que queremos ejecutar para diferentes valores de N, es decir, hasta qué cantidad de ceros queremos ejecutar, si le decimos 1 lo hace para N = 10 solo, si decimos 6 hace desde 10 hasta 1000000.

El tiempo se empieza a medir justo antes de la declaración de variables y se finaliza justo al finalizar los bucles. Este tiempo se suma a una variable global y cuando se finalizan todas las ejecuciones se hace la media como la suma total entre la cantidad de iteraciones realizadas y se muestra por pantalla.

En ninguno de los programas se realiza un calentamiento de la caché, de forma que las primeras ejecuciones del código, para N bajos, cuentan con el añadido de rellenar la caché.

Código ensamblador

El código ensamblador generado para los lazos de ambas partes es el siguiente:

Sin optimizar:

```
        movl    $0, -144(%rbp)
        jmp     .L11
.L17:
        movl    $0, -140(%rbp)
        jmp     .L12
.L16:
        movl    -148(%rbp), %eax
        movl    %eax, %edx
        shrl    $31, %edx
        addl    %edx, %eax
        sarl    %eax
        cmpl    %eax, -140(%rbp)
        jne     .L13
        movq    -80(%rbp), %rax
        movl    -140(%rbp), %edx
        movslq  %edx, %rdx
        pxor    %xmm0, %xmm0
        movss   %xmm0, (%rax,%rdx,4)
        jmp     .L14
.L13:
        movl    -148(%rbp), %eax
        subl    $1, %eax
        cmpl    %eax, -140(%rbp)
        jne     .L15
        movl    -148(%rbp), %eax
        subl    $1, %eax
        cvtsi2ss        %eax, %xmm0
        movq    -80(%rbp), %rax
        movl    -140(%rbp), %edx
        movslq  %edx, %rdx
        movss   %xmm0, (%rax,%rdx,4)
        jmp     .L14
.L15:
        movq    -80(%rbp), %rax
        movl    -140(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm1
        movq    -64(%rbp), %rax
        movl    -140(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm0
        addss   %xmm1, %xmm0
        movq    -80(%rbp), %rax
        movl    -140(%rbp), %edx
        movslq  %edx, %rdx
        movss   %xmm0, (%rax,%rdx,4)
```

Optimizado:

```
        movl    $0, -148(%rbp)
        jmp     .L10
.L15:
        movl    $0, -144(%rbp)
        jmp     .L11
.L12:
        movq    -80(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm1
        movq    -64(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm0
        addss   %xmm1, %xmm0
        movq    -80(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   %xmm0, (%rax,%rdx,4)
        addl    $1, -144(%rbp)
.L11:
        movl    -124(%rbp), %eax
        movl    %eax, %edx
        shrl    $31, %edx
        addl    %edx, %eax
        sarl    %eax
        cmpl    %eax, -144(%rbp)
        jl      .L12
        movl    -124(%rbp), %eax
        movl    %eax, %edx
        shrl    $31, %edx
        addl    %edx, %eax
        sarl    %eax
        movl    %eax, %edx
        movq    -80(%rbp), %rax
        movslq  %edx, %rdx
        pxor    %xmm0, %xmm0
        movss   %xmm0, (%rax,%rdx,4)
        movl    -124(%rbp), %eax
        movl    %eax, %edx
        shrl    $31, %edx
        addl    %edx, %eax
        sarl    %eax
        addl    $1, %eax
        movl    %eax, -144(%rbp)
        jmp     .L13
.L14:
        movq    -80(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm1
        movq    -64(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   (%rax,%rdx,4), %xmm0
        addss   %xmm1, %xmm0
        movq    -80(%rbp), %rax
        movl    -144(%rbp), %edx
        movslq  %edx, %rdx
        movss   %xmm0, (%rax,%rdx,4)
        addl    $1, -144(%rbp)
.L13:
        movl    -124(%rbp), %eax
        subl    $1, %eax
        cmpl    %eax, -144(%rbp)
        jl      .L14
        movl    -124(%rbp), %eax
        leal    -1(%rax), %ecx
        movl    -124(%rbp), %eax
        leal    -1(%rax), %edx
        cvtsi2ss        %ecx, %xmm0
        movq    -80(%rbp), %rax
        movslq  %edx, %rdx
        movss   %xmm0, (%rax,%rdx,4)
        addl    $1, -148(%rbp)
```

Como podemos observar a primera vista la diferencia en la cantidad de instrucciones es evidente.

Podemos ver como hay tres bloques dedicados a cada uno de los condicionales: L16 corresponde a si i vale N/2 y la asignación en caso afirmativo; L13 corresponde a si i vale N-1 y su asignación en caso afirmativo; y L15 corresponde al else y la asignación por defecto. Las últimas instrucciones del primer bloque corresponden al bucle de ITER y las instrucciones del bloque L17 al bucle de N.

En el caso del código optimizado podemos ver como las primeras instrucciones, L15, L12 y L11 están dedicados al primer bucle hasta N/2; L11 que contiene la asignación sobre la posicion N/2 de x a 0; el segundo bucle que está en L11, L14 y L13; y L13 que contiene la asignación de valor sobre la última posición de x.

Para poder identificar correctamente las partes del código he empleado la herramienta https://godbolt.org/.

Resultados

Ambos programas han sido ejecutado con la opción de no optimización de gcc, mediante *“gcc -O0 codigo.c”.*

Para cada código se han realizado 6 ejecuciones con diferentes tamaños de N cada una, pero el valor de ITER siempre el mismo (10000) y para cada N se han realizado 10 ejecuciones, siendo el tiempo la media de tiempos en las 10 ejecuciones.

Resultado código sin optimizar
N = 10, ITER = 10000, iteraciones = 10, media tiempo = 0.002612 seg
N = 100, ITER = 10000, iteraciones = 10, media tiempo = 0.020941 seg
N = 1000, ITER = 10000, iteraciones = 10, media tiempo = 0.191364 seg
N = 10000, ITER = 10000, iteraciones = 10, media tiempo = 1.897380 seg
N = 100000, ITER = 10000, iteraciones = 10, media tiempo = 18.497084 seg
N = 1000000, ITER = 10000, iteraciones = 10, media tiempo = 188.302802 seg

Resultados código optimizado:
N = 10, ITER = 10000, iteraciones = 10, media tiempo = 0.001173 seg
N = 100, ITER = 10000, iteraciones = 10, media tiempo = 0.015637 seg
N = 1000, ITER = 10000, iteraciones = 10, media tiempo = 0.081908 seg
N = 10000, ITER = 10000, iteraciones = 10, media tiempo = 0.978715 seg
N = 100000, ITER = 10000, iteraciones = 10, media tiempo = 7.947154 seg
N = 1000000, ITER = 10000, iteraciones = 10, media tiempo = 72.658158 seg

Como podemos observar el tiempo es para todos los valores de N menor con optimización que sin ella. Podemos ver como para casi todos los tamaños, a excepción de N con valor 100, el tiempo empleado por el código optimizado es menos de la mitad que el tiempo empleado por el código sin optimizar.

Estos resultados confirman que es una muy buena optimización.

Conclusiones

Tal y como hemos visto durante el informe realizar este tipo de optimizaciones en el código pueden suponer hasta la mitad de tiempo de ejecución, más allá de que ambas formas de recorrer bucles sean perfectamente válidas. Esto demuestra que a la hora de programar se debe tener en cuenta este tipo de cosas, no solo generar código funcional sino eficiente también.

Referencias

(1) Loop Splitting (2 de diciembre de 2021). Wikipedia: https://en.wikipedia.org/w/index.php?title=Loop_splitting&oldid=1050438254