Optimización código

Índice

- Opciones de compilación código.
  - Niveles de optimización.
- Profiling
- Aliasing
- Intrínsecos
  - Add
  - Alligment
- Pragmas
  - Trip count
  - Desenrrollo
- _nassert
- Keywords
- Declaración ASM

Qué significa código óptimo

¿ Qué significa?

Qué significa código óptimo

Depende de tu objetivo:

- Tener algoritmo que funcione a tiempo real.
- Uso de CPU al mínimo.
- Que el jefe te diga que ya es suficiente.

Realtime vs CPU min

Tiempo real:

- Que la aplicación se ejecute correctamente en el momento. Requiere dotar al compilador de las opciones correctas (fácil).

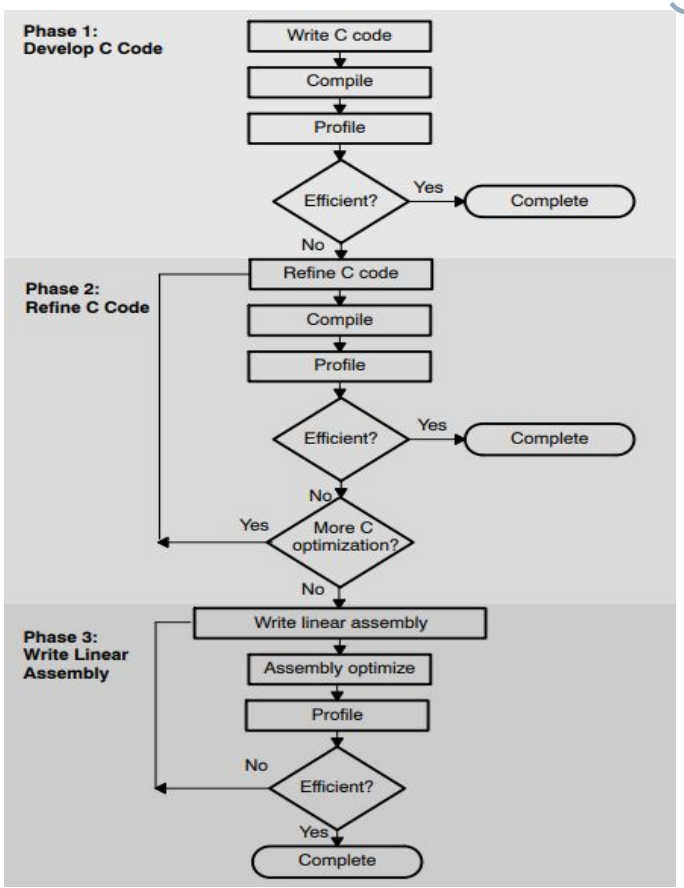
CPU mínimo:

- Esto también requiere conocimientos de arquitecturas para reducir ciclos de reloj (requiere más tiempo).

Optimización

Es el proceso de refinamiento de código en búsqueda de ejecutarse rápido y en pocos ciclos y lograr cumplir el objetivo que haya sido propuesto.

Aumentar rendimiento código



Debug vs optimización

Debug:

- Revisar tu código y mejorarlo lógicamente.

Optimización:

- Usar técnicas que te permiten optimizar el código a la hora de la compilación.

Consejos básicos para código en C

- Usar mínima complejidad de código.
- No llamar a funciones en los bucles
- Bucles pequeños
- Generar tests, probar el código

Ámbitos para lograr optimización programando DSP

- Opciones del compilador
- Instrucciones y pragmas que facilitan
- Keywords especiales
- Técnicas generales de programación

Opciones del compilador: Niveles de optimización

Opciones del compilador

- Se puede decir al compilador que tome partido de diferentes formas en pro de optimizar el código.
- Hay varios niveles en los que actuar.
- Elegir muy bien el nivel de optimización y no mentir al compilador para que no haya problemas
- Usar diferentes niveles para arreglar el problema.
- El pm (ó -o3) es el más general.

Nivel --opt_level=0 or -O0. Optimización del uso del registro

- Realiza la simplificación del gráfico de flujo de control.
- Asigna variables a registros.
- Realiza rotación de bucle.
- Elimina el código no utilizado.
- Simplifica expresiones y declaraciones.
- Expande las llamadas a funciones declaradas en línea.

Nivel --opt_level=1 ó -O1. Optimización en nivel de bloque.

Realiza todas las optimizaciones --opt_level = 0 (-O0), más:

- Realiza copia local / propagación constante.
- Elimina las asignaciones no utilizadas.
- Elimina expresiones comunes locales.

Nivel --opt_level=2 or -O2. Optimización global

Realiza todas las optimizaciones --opt_level = 1 (-O1), más:

- Realiza pipeline.
- Realiza optimizaciones de bucle.
- Elimina subexpresiones comunes globales.
- Elimina asignaciones globales no utilizadas.
- Convierte referencias de matriz en bucles a forma de puntero incrementado.
- Realiza desenrollado de bucles.

Nivel --opt_level=2 or -O2. Optimización global

El optimizador usa --opt_level = 2 (-O2) como predeterminado si usa --opt_level (-O) sin una optimización nivel.

Nivel --opt_level=3 or -O3. Optimización nivel de archivo

Realiza todas las optimizaciones --opt_level = 2 (-O2), más:

- Elimina todas las funciones que nunca se llaman.
- Simplifica funciones con valores de retorno que nunca se utilizan.
- Llamadas en línea a pequeñas funciones.

Nivel --opt_level=3 or -O3. Optimización nivel de archivo

- Reordena las declaraciones de funciones, los atributos de las funciones llamadas se conocen cuando el llamador es optimizado.
- Propaga argumentos en cuerpos de funciones cuando todas las llamadas pasan el mismo valor en el mismo posición del argumento.
- Identifica características variables en archivo.

Profiling

Profiling

En aplicaciones grandes tiene más sentido optimizar las secciones de código primero. Para ello:

- Usar opción -g al compilar.
- También se puede obtener información de recuento de ciclos para una función o región de código con el simulador independiente, incruste la función clock() en su código C.

Aliasing

Aliasing

El problema del aliasing sucede cuando 2 o más punteros apuntan a la misma dirección de memoria. Esto puede provocar errores .

Soluciones Aliasing

1. El compilador suele solucionar la mayoría de los casos, aunque no lo optimiza.
2. Indicas el nivel de optimización (-pm -o3) y así visualizas el conjunto.
3. Forzar el no bad aliasing(-mt). Con esto le dices al programa que no hay aliasing. Si por lo que sea va muy lento porque no estás seguro si eso se está cumpliendo, sería quitar el aliasing y probar
4. Restrict keyword. Similar que el -mt pero en nivel de base.

Intrínsecos

Intrínsecos

El compilador C6000 proporciona funciones especiales intrínsecas que se asignan directamente a instrucciones para optimizar el código C con rapidez. Esto son llamados funciones “Intrinsics”. Los intrínsecos se especifican con un subrayado inicial (_) y se accede llamándolos mientras llama a una función.

Apoyado en esos intrínsecos se reduce ciclos de reloj a operaciones que programadas tardarían más.

Lista de intrínsecos

Verse la lista de intrínsecos en el documento “TMS320C6000 Programmer’s Guide” a partir de hoja 50.

Ejemplo con _add2

Suma la mitad superior e inferior de src1 a la mitad superior e inferior de src2 y devuelve el resultado. Cualquier desbordamiento al sumar la mitad inferior no afecta a la mitad superior:

```
void vecsum4(short *restrict sum, restrict short *in1, restrict short *in2, unsigned int N)
{
        int i;
        const int *restrict i_in1 = (const int *)in1;
        const int *restrict i_in2 = (const int *)in2;
        int *restrict i_sum = (int *)sum;
        for (i = 0; i < (N/2); i++)
                i_sum[i] = _add2(i_in1[i], i_in2[i]);
}
```

Alignment

Data types C6000

Type	Bits	Bytes
char	8	1
short	16	2
int	32	4
long	32	4
long long	64	8
float	32	4

...

Data types C6000

A la hora de usar los intrínsecos tener muy en cuenta los tamaños del dato. Puede ser de mucha utilidad para jugar con datos alineados.

Intrínseco _amem vs _mem

Entre los intrínsecos más utilizados tenemos _amem y _mem.

1. Te permiten alinear por bytes para poder realizar operaciones (sumas, multiplicaciones) más rápido.
2. Pueden ser alineaciones de 2, 4 o 8.
3. _mem es más general. Funcionará si la variable está alineada previamente o no. _amem es más estricto: requiere que la variable esté alineada (aunque presumiblemente es un poco más eficiente). Por lo tanto, usa _mem a menos que pueda garantizar que la variable siempre esté alineada.

Pragmas

Pragmas

- Son directivas que le dicen al compilador cómo tratar una determinada función, objeto o sección de código.
- Se escriben indicando en el lugar donde la queremos colocar. #pragma
- Decirle información concreta al compilador a mejorar su rendimiento

Lista de pragmas

Verse la lista de pragmas en el documento “TMS320C6000 Optimizing Compiler v 7.4 User's Guide” a partir de hoja 157.

#pragma DATA_ALIGN

Alinea variables. Fuerza para que haya determinado número bytes por cada tantos para así asegurarnos un correcto uso de la memoria. Tiene que ser múltiplo de 2.

Trip count

Trip count

El recuento de viajes o trip count es el número mínimo de veces que se ejecuta un bucle.

Si el compilador no puede determinar que un ciclo siempre se ejecutará para el mínimo de viajes, entonces genera un ciclo redundante, sin pipelining.

Trip count

Si el compilador puede garantizar que se ejecutarán al menos n iteraciones de bucle, entonces n es el recuento mínimo de viajes conocido.

A veces, el compilador puede determinar esta información automáticamente. Alternativamente, puede proporcionar esta información utilizando el pragma MUST_ITERATE y PROB_ITERATE.

Pragma MUST_ITERATE

Esta directiva especifica ciertas propiedades al bucle, con lo que el compilador va a tardar menos. Eres tú el que vas a garantizar que estas propiedades son ciertas.

Garantizas el número de ejecuciones para que el compilador sea más eficiente.

#pragma PROB_ITERATE

Como la anterior MUST_ITERATE pero sus parámetros son más restrictivos.

_nassert (intrínsec)

_nassert () es una función que he estado usando para mejorar la plataforma C64X + del compilador DSP. Ayuda a decirle al compilador cómo se alinean los accesos a la memoria diciéndole si algo sucede. Permitiéndole al compilador que pueda generar un mejor código y programación.

Por ejemplo, **_nassert(((int)(a) & 0x3) == 0))** permite decirle al compilador si el puntero a está alineado con el límite de la palabra

Eliminar bucles redundantes (1)

A veces, el compilador no puede determinar si el bucle se ejecuta más que el count de trips que tiene asignado seguro. Por lo tanto, el compilador generará dos versiones del bucle:

- Una versión con el count de trips mínimos seguros.
- Otra con una cuenta de viajes mayor o igual a la segura.

Esto genera bucles redundantes y afecta al rendimiento.

Eliminar bucles redundantes (2)

Para indicarle al compilador que no deseamos dos versiones del bucle, se puede usar la opción −ms0 o −ms1. El compilador generará el pipeline del bucle sólo si puede probar que el trip count es siempre igual o mayor que el recuento mínimo efectivo.

Para ayudar al compilador a generar solo un pipeline se puede usar el pragma MUST_ITERATE y / o la opción −pm para ayudar al compilador a determinar el trip count conocido.

Indagando más en MUST_ITERATE (2)

Ejecución de un bucle al menos 30 veces.

#pragma MUST_ITERATE (30);

Indagando más en MUST_ITERATE (1)

Ejecución de un bucle 30 veces exactamente.

#pragma MUST_ITERATE (30, 30);

Indagando más en MUST_ITERATE (3)

El trip count se ejecutará un múltiplo de 4 veces.

#pragma MUST_ITERATE (,, 4);

Indagando más en MUST_ITERATE (4)

Se puede combinar las diferentes opciones anteriores.

#pragma MUST_ITERATE (8, 48, 8);

Desenrrollo bucles

Desenrrollar bucles

El desenrrollo consiste en expandir pequeños bucles, escribiendo cada iteración resultante en el código y por tanto evitando escribir un for.

El desenrrollo aumenta el número de instrucciones disponible para ejecutar en paralelo. Por tanto es beneficioso usar esta técnica cuando las operaciones en una sola iteración no está haciendo uso de todos los recursos.

Formas de desenrrollarlo

Hay tres formas de desenrollar el bucle:

1. El compilador puede desenrollar automáticamente el ciclo.
2. Puede sugerir que el compilador desenrolle el bucle usando el pragma UNROLL.
3. Puede desenrollar el código uno mismo.

Mejora bucles

Para mejorar los bucles tenemos diferentes áreas que podemos tratar de una forma que nos ayude a optimizar el código:

- Trip count.
- Bucles redundantes.
- Desenrollado de bucles.
- Ejecución especulativa.

Keywords

Keywords

- El compilador C6000 C / C ++ admite las palabras clave estándar const, register, restrict y volatile. Además, el compilador C / C ++ extiende el lenguaje C / C ++ mediante el soporte de las palabras clave cregister, interrupt, near y far.

- Mediante las keywords podemos decirles indicarles al compilador ciertas caracterísitcas de nuestro código que hace que mejores la eficiencia.

Keywords: Restrict

- Para ayudar al compilador a determinar las dependencias de la memoria, puede calificar un puntero, una referencia o una matriz con la palabra clave restrict.
- Su uso representa una garantía por parte uno mismo, el programador, de que dentro del alcance de la declaración del puntero, solo se puede acceder al objeto apuntado por ese puntero.
- Cualquier violación de esta garantía deja el programa indefinido.

Keywords: Restrict

- Esta práctica ayuda al compilador a optimizar ciertas secciones de código porque la información de alias se puede determinar más fácilmente.

Keywords: Const

- Const especifica que el valor de una variable es constante e indica al compilador que evite que el programador lo modifique
- En resumen, variable que solo se lee, no se reescribe.

Declaración ASM

El compilador C / C++ puede incrustar instrucciones o directivas en lenguaje ensamblador directamente en la salida del compilador en lenguaje ensamblador. Esta capacidad es una extensión del lenguaje C / C++, la declaración asm. La declaración asm (o __asm) proporciona acceso a funciones de hardware que C / C ++ no puede proporcionar.