调试 Blackfin®处理器编译 C 源代码

作者：DSP Tools Compiler Group *May 26, 2003*

引言

本文档利用 VisualDSP++™版本 2.0，为通过 Blackfin®系列处理器的 C/C++编译器获得优化的程序代码实现功能，提供了一些技术指导。

使用优化器

编译优化和没有优化的 C 代码在性能方面有巨大的差别，某些情况下，优化后的程序代码运行效率可能提高 10 倍甚至 20 倍，因此，在测量算法性能或者将代码移植成产品之前，都应该尝试进行优化。需要说明的是，默认编译器设置是非优化的，这样做是为了便于程序员通过其原始的程序代码诊断错误。

由于 C 程序是直接方式编写的，因此编译器中的优化器用来将 C 程序生成执行效率高的代码。调试程序的一个基本策略是以某种方式给出算法，使优化器能极好的可视化所有操作与数据，这样就有很大的自由度去对代码实施安全操作。需要说明的是，以后的版本将更好的增强优化器，用更简洁的表示算法，以获得增强带来的益处。

使用统计性能评测器

调试源程序首先应理解应用中哪部分是热点。VisualDSP++中提供的统计性能分析是寻找这些热点的有效方法。

如果对应用程序不熟悉，可执行带有诊断的编译，不用优化运行，即可得到一些与 C 程序直接相关的结果，如果完全优化编译了应用程序，并得到了直接与汇编代码相关的统计结果，则可以更准确的查看应用程序。可能出现的唯一问题是如何将汇编语句行与原始的源程序相关联。在链接的时候，不要去掉函数名，有了函数名，就可以滚动汇编窗口，定位热点。在十分复杂的代码中，除非没有展开源程序，都可通过循环计数定位确切的源程序行。在.s 文件中查看行数，注意编译器汇编可能移动程序代码。

字符型	8位带符号整数
无符号字符型	8位无符号整数
短字	16位带符号整数
无符号短字	16位无符号整数
整型	32位带符号整数
无符号整型	32位无符号整数
长整型	32位带符号整数
无符号长整型	32位无符号整数

表1：定点数据类型(原始算法)

编译器直接支持 10 种格式的数据类型，如表 1 和表 2 所示。双精度型等效于 Blackfin 处理器的浮点类型，处理器硬件不直接支持 64 位数据类型。

浮点	32位浮点
双精度	32位浮点

表2：浮点数据类型(仿真算法)

小数数据类型可表示为短字型或者整型。这些数据类型的操作最好通过指令来完成，将在下面的章节中讨论。

避免浮点/双精度运算

浮点算术操作通过子程序库实现，由于循环中的浮点算法操作阻止优化器使用硬件循环，因此其操作比整数操作慢得多。

在循环中避免整数除法

处理器硬件不直接支持 32 位整数除法，所以对于整型变量的除法和取余操作是多周期操作。如果除数已知，编译器将把除以 2 的幂型整数除法转换成右移操作。

如果编译器必须实施完整的除法操作，它将调用库函数，除了多周期操作外，对于任何与除法有关的循环，都将阻止优化器使用硬件循环。因此，如果可能，尽量不要在循环中使用除法和取余操作。

变址矩阵与指针

C 语言规则允许以两种方式对矩阵中数据的编程访问：一是通过不变的基地址指针变址，一是增加指针的值。以下加法的两种实现版本说明了这两种类型：

```
void va_ind( short a[], short b[],
             short out[], int n)
{
  int i;
  for (i = 0; i < n; ++i)
    out[i] = a[i] + b[i];
}
```

列表 1：变址矩阵

```
void va_ptr( short a[], short b[],
             short out[], int n)
{
  int i;
  short *pout = out, *pa = a, *pb = b;
  for (i = 0; i < n; ++i)
    *pout++ = *pa++ + *pb++;
}
```

列表 2：指针

通常认为选择哪种类型对生成的代码不产生任何差别，但某些情况下它确实有差别。算法的一个版本可能比另外一个能产生更优化的代码，但并不总是该版本一直要优越一些，生成的代码总受到周围程序代码的影响，这就是会出现差异的原因。在编译器分析过程中，指针类型引入额外的变量，与周围的程序代码竞争资源；另一方面，对于矩阵访问，编译器必须先转换成指针，而这一过程有时并不如人工转换有效。

最好的策略就是从数组符号开始，这看起来并不是一个令人满意的使用指针的尝试。而在重要循环的外部，使用变址访问风格，这样更容易理解。

使用-ipa 切换

为了保证最好的性能，优化器通常需要知道某些由外部工作的子程序确定的事情。特别是它要帮助了解指针参数的对准和数值，以及循环边界的值。编译器的-ipa 切换可以使能内部过程化分析(IPA)，以获得这些信息。在 IDDE 环境中的

Project 选项菜单下，有个 Project Options 对话框，在 Compile 表中，选中 Interprocedural Optimization 对话框即打开该功能。

当使用了该选项，在链接阶段可能重新调用编译器，并利用上次编译获得的额外信息再重新编译程序。

由于该选项只在链接时工作，如果在编译时使用了-S 选项，则无法看到-ipa 选项的效果。要查看汇编器文件，在 Project Options 对话框中的 Compile 表中，在 Additional Options 文本框中输入-save-temps，在编译完程序后，查看生成的.s 文件即可。

以下很多建议都假定使用了-ipa 选项。

静态初始化常量

内部过程化分析能够识别只有一个值的变量，并利用常量代替，这可以使能更好的优化。如果这样做，该变量在整个程序中都只能有一个值。

```
#include <stdio.h>
static int val = 3; // initialized
                    // once
void init() {
}
void func() {
  printf("val %d",val);
}
int main() {
  init();
  func();
}
```

列表 3：优化(IPA 已知 VAL 值为 3)

正如所有全局变量为默认值一样，如果变量静态初始化为零，随后在程序中的另外一点分配了其他某个值，则分析器可以看到两个值，该变量就不被当作具有常量值。

```
#include <stdio.h>
static int val; // initialized to zero
void init() {
  val = 3; // re-assigned
}
void func() {
  printf("val %d",val);
}
int main() {
  init();
  func();
}
```

列表 4：未优化(IPA 未将 VAL 看作常量)

数据字对齐

为了最有效的使用硬件，必须对齐数据输入。在多数算法中，用于计算的数据访问的平衡也是如此，为了充分使用硬件，取出的数据必须是 32 位宽度。

虽然 Blackfin 的结构支持字节寻址，但硬件上仍要求对存储器的引用在本质上是对齐的，因此，16 位的参考地址必须在偶数位置，即 32 位的字对齐地址。因此，为了生成高效的程序代码，应确保数据按 32 位对齐。

编译器能帮助建立阵列数据的对齐。堆栈帧也是按字对齐的。不管顶层数据阵列的数据类型如何，他们总是字对齐的。

如果编写的程序只是将某阵列中第一个元素的地址作为参数传递，对输入阵列循环的过程，从元素零开始，一次一个元素，则内部过程分析就能够建立，这种对齐也就适合 32 位访问。

在内循环处理多维数组的单行数据时，应确认每行数据都从字边界开始，如果不是，应尽可能插入多余数据确保如此。

循环向导

附录 A 回顾了编译器如何转换循环以生成高效的代码，并说明了“循环展开”优化技术。

不要自己展开循环

循环展开不仅使程序难于阅读，也妨碍了程序优化。为了能自动使用宽的负载和两个累加器，编译器必须具有自己展开循环的能力。

```
void va1(short a[], short b[],
         short c[], int n)
{
  int i;
  for (i = 0; i < n; ++i) {
    c[i] = b[i] + a[i];
  }
}
```

列表 5：优化(编译器展开，使用两个计算块)

```
void va2(short a[], short b[],
         short c[], int n)
{
  short xa, xb, xc, ya, yb, yc;
  int i;
  for (i = 0; i < n; i+=2) {
    xb = b[i]; yb = b[i+1];
    xa = a[i]; ya = a[i+1];
    xc = xa + xb; yc = ya + yb;
    c[i] = xc; c[i+1] = yc;
  }
}
```

列表 6：未优化(编译器脱离 16 位载入数)

在这个例子中，内部过程分析能确定 a,b,c 的初始值都以 32 位边界对齐，且 n 是 2 的倍数的情况，第一个版本的循环运行速度几乎比第二个快三倍。

避免循环加载的相关性

循环加载的相关性是指在一个给定的循环重复过程中，若不知道前一次循环计算出的值，就无法完成本次计算。当一个循环具有相关性时，编译器就不能重复该循环。

如果在某个循环中使用的标量在使用之前没有定义，也会引起这种相关性。

```
for (i = 0; i < n; ++i)
  x = a[i] - x;
```

列表 7：未优化(标量相关性)

在出现称为约减型的标量相关性时，优化器可以对循环递归重新排序。这些循环利用相关的且可交换的运算符将矢量值简化为标量值，最常见的例子就是乘法和累加运算。

```
for (i = 0; i < n; ++i)
  x = x + a[i] * b[i];
```

列表 8：优化(一种约减)

在第一个实例中，标量相关性是相减操作。当循环不按顺序执行，则会生成不同的结果，变量 x 就是以这种方式变化。作为比较，在第二个实例中，加操作的特性就意味着编译器可以按任意顺序执行操作，但总能得出相同的结果。

不用手动方式循环

DSP 代码中，常常以手动方式进行循环，这样做是为了在计算当前循环的同时，能完成前一次循环和后一次循环中的数据加载和存储操作。该技术引入了循环加载的相关性，阻止了编译器有效的重排程序代码。最好是给编译器一个“规范化”的版本，将循环操作留给编译器去完成。

```
int ss(short *a, short *b, int n) {
  short ta, tb;
  int sum = 0;
  int i = 0;
  ta = a[i]; tb = b[i];
  for (i = 1; i < n; i++) {
    sum += ta + tb;
    ta = a[i]; tb = b[i];
  }
  sum += ta + tb;
  return sum;
}
```

列表 9：未优化(手动循环)

通过旋转循环，可以加入标量值 ta 和 tb，并引入循环加载的相关性，这样可以防止编译器实施并行迭代，而优化器本身能够完成这种循环旋转。

```
int ss(short *a, short *b, int n) {
  short sum = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum += a[i] + b[i];
  }
  return sum;
}
```

列表 10：优化(由编译器完成旋转)

避免在循环中进行数组写操作

对数组元素进行写操作还会引入其他相关性。在下面的循环中，优化器不能确定对 a 的载入值究竟是来自上次循环中定义的，还是在随后循环中将重新写入的数值。

```
for (i = 0; i < n; ++i)
  a[i] = b[i] * a[c[i]];
```

列表 11：未优化(数组相关性)

```
for (i = 0; i < n; ++i)
  a[i+4] = b[i] * a[i];
```

列表 12：优化(感应变量)

优化器可以解析访问模式，该模式中地址是每次循环中以固定量改变的表达式，即所谓的“感应变量”。

避免混淆

```
void fn(short a[], short b[], int n)
{
  for (i = 0; i < n; ++i)
    a[i] = b[i];
}
```

列表 13：未优化(可能的别名混淆)

表面上看，以上这个循环看起来不具有任何循环加载的相关性，但 a 和 b 都是参数，虽然它们通过带有[ ]的符号进行了声明，实际上它们仅是指针，而且可能指向同一数组。当通过这两个指针能指向同一数组时，我们就说它们可能会相互混淆。

如果使能了-ipa 选项，编译器就能够找到 fn 的调用地址，而且还能确定它们是否指向同一数组。

即使用了-ipa 选项，也十分容易造成明显的别名混淆。过程分析器以将指针与变量数组关联起来的方式进行工作，而这些变量组可能涉及到程序中的某些点。为了简化分析，可能没有考虑到控制流程，若发现两个指针所指的变量数组存在交叉引用，则会认为两个指针都指向两个变量数组的联合体。

如果上面的函数 fn 被两个全局数组作为参数调用，针对以下两种情况，则内部过程分析将得到以下结果：

Case 1:

fn(glob1, glob2, N); 数组没有混合使用a和b没有
fn(glob1, glob2, N); 混淆(优化)

Case 2:

fn(glob1, glob2, N); 数组没有混合使用a和b没有
fn(glob3, glob4, N); 混淆(优化)

Case 3:

fn(glob1, glob2, N); 数组混合使用，a和b
fn(glob3, glob1, N); 可能混淆(非优化)

第三种情况说明了 IPA 在确定是否存在混淆的风险时，要立刻考虑所有调用的联合，而不是对每个调用进行单独考虑。若对每个调用都单独考虑，IPA 就必须进行流控制，而排列的数目将会使编辑时间长得无法实施。

尽量在内循环中做更多的工作

优化器主要着眼于内循环，因为大部分程序的大多数时间都消耗在这里。对于优化而言，要使程序的循环体运行更快，应该在循环之前和之后减慢程序，这是一种很好的开销。因此，应确保算法的大部分时间都用于内循环。否则，优化操作事实上还可能使其运行减慢。

一种有用的技术就是“循环转换”。若一个嵌套循环的外循环运行次数很多而内循环运行次数较少，重写循环就有可能减少外循环的循环次数。

循环中避免使用条件代码

若一个循环中包含条件代码，当分支结果和编译器的预测存在很大分歧时，将可能存在很大代价。在某些情况下，编译器能够把 if-else 和?:结构转换为条件运行，而在其他情况下，需要重新配置表达式评估整个循环的外部。然而，对于重要的循环，最好写成线性代码。

保持循环简短

为了使编译器效率最高，循环体应尽可能简短。大的循环体常常更复杂且难于优化。

此外，循环还可能需要将寄存器数据存入存储器中，这也会导致代码密度和执行性能的下降。

不要在循环中使用函数调用

如果循环中包含一个函数调用，编译器就不能生成硬件循环，这是因为保存和重新载入硬件循环的文本都有代价。除了明显的函数调用，如 printf()外，一些操作仍会阻碍硬件循环的生成：整数除法和取余，浮点运算，以及整型和浮点型数据之间的转换，这些操作，都存在隐含支持程序的调用。

用整数作为循环控制变量和数组索引

对于循环控制变量和数组索引，最好用整型而不用短整型。在标准 C 中声明了短整型在进行计算前必须扩展成整型大小，然后再截断到原来的短整型大小。

编译器通常能够处理短整型循环计数器，并能检测零开销循环和指针“感应变量”。然而，这也会增大编译器的工作负担，偶尔还会生成没有充分优化的程序代码。

循环编译指示和辅助矢量化

```
void copy(short *a, short *b)
{
  int i;
  for (i=0; i<100; i++)
    a[i] = b[i];
}
```

列表 14：未优化(无编译指示)

如果调用列表 14 中的函数 copy 两次，比如先调用 copy(x, y)然后再调用 copy(y, z)，如前文所述，内部过程分析不能说明 a 永远不会与 b 混淆。因此，该循环就包含了循环加载的相关性，且不能矢量化操作。这种情况下，一种解决方法就是利用 vector_for 编译指示，告诉编译器一次循环过程中的计算不依赖前一次循环中计算出的数据。

以下程序代码使用 vector_for 编译指示，允许循环并行完成两次重复运行。

```
void copy(short *a, short *b)
{
  int i;
  #pragma vector_for
  for (i=0; i<100; i++)
    a[i] = b[i];
}
```

列表 15：优化(利用编译指示)

需要说明的是，该编译指示不能强制编译器将循环进行矢量化操作，编译器将检测循环的各项属性。若编译器认为该循环不安全，或不能找到进行矢量化操作所需的各种属性，它就不会将该循环矢量化。编译只是向编译器确保不存在循环加载相关性，但可能还有其他循环属性会阻碍循环的矢量化操作。

当矢量化操作是无法实现(比如，如果数组 a 是字边界对齐，而 b 不是)，此时由 vector_for 提供的判断提示信息在辅助其他优化选项时仍然可能用到。

const 数据是常数

默认情况下，编译器都假定指针指向的数据为不变的 const 数据。因此，告知编译器两个数组 a 和 b 没有发生重载的另外一种方法就是利用 const 关键字。

列表 16 中给出的例子与 vector_for 编译指示具有类似的作用。事实上，采用 const 更好，这是因为在矢量化操作后，仍允许优化器旋转循环，而它要求知道逐次非相关的循环迭代恰好是不相邻的迭代，而以后循环迭代更为不同。

```
void copy(short *a, const short *b)
{
  int i;
  for (i=0; i<100; i++)
    a[i] = b[i];
}
```

列表 16：const 关键字的用法

在 C 语言中，用 const 语句改变指针指向的常量数据这一编程习惯虽然不好，但却是合法的，即使如此，仍应避免这种情况。默认情况下，编译器生成的代码都假定 const 数据不会发生变化。然而，若有一程序通过指针修改 const 数据，利用编译时间标记-const-read-write，也可以生成正确的代码。

分数型数据

分数型数据用 16 位和 32 位整数表示，有两种操作方式。推荐的方式是使用内在的值，这样可以最大限度地控制数据。考虑以下分数标量积，可编程如下：

```
int sp(short *a, short *b)
{
  int i;
  int sum=0;
  for (i=0; i<100; i++) {
    sum += ((a[i]*b[i]) >> 15);
  }
  return sum;
}
```

列表 17：未优化(使用移位)

然而，这对于优化器而言却存在很多问题。通常，这里生成的代码是一次乘法运算，接着是一次移位，最后是一次累加。但是，Blackfin 处理器提供了分数乘法累加指令，可以在单周期内完成所有这些操作。而且，它还可以并行执行两条这样的指令。

编译器可以识别该术语，并能确认在 DSP 系统中，使用这种优化是为了使算法达到饱和。乘法/移位操作被饱和的分数乘法代替。不需要饱和处理时，通过使用-no_int_to_fract 选项，也可以禁止这种转换。

而这还只是一种简单情况，在更复杂的情况下，乘法可能与移位操作相距较远，就不存在编译器能检测到使用分数乘法的可能性。推荐的编程风格就是使用内在的风格。在以下的例子中，add_fr1x32()和 mult_fr1x32 分别用于实现 32 位分数数据的加法和乘法。

```
#include <fract.h>
fract32 sp(fract16 *a, fract16 *b) {
  int i;
  fract32 sum=0;
  for (i=0; i<100; i++) {
    sum = add_fr1x32(sum,
    mult_fr1x32(a[i],b[i]));
  }
  return sum;
}
```

列表 18：优化(使用内在值)

Blackfin 处理器 C/C++编译器手册给出了全部的分数操作列表，同时还给出了其他可用内在值的说明。内在函数提供单 16 位或 32 位数值的常用操作。在这种情况下，编译器可以确定在何时循环可以被矢量化操作，并生成 2x16 个操作。正如将循环旋转留给编译器处理更好，内在值将配对操作对留给编译器完成最好。

若可能将数组放入不同存储区

Blackfin 处理器能够在单个指令行上支持两种存储器操作。然而，只有当两个地址在不同的存储块中时，该操作才能在单周期内完成。如果同时对同一存储块访问，将会插入一次空操作。点乘就是范例(如前面部分所示)。

由于每个周期都从数组 a 和 b 载入数据，确保这些数组位于不同的存储块中是十分有益的。例如以下.LDF 文件的 MEMORY 部分就定义在两个不同的存储区。

```
MEMORY {
  BANK_A1 {
   TYPE(RAM) WIDTH(8)
   START(0xFF900000) END(0xFF900FFF)
  }
  BANK_A2 {
   TYPE(RAM) WIDTH(8)
   START(0xFF901000) END(0xFF901FFF)
  }
}
```

列表 19：LDF 存储配置

然后配置 SECTIONS 部分，告诉链接器将数据部分放入指定的的存储区块，如下所示。

```
SECTIONS {
  bank_a1 {
    INPUT_SECTION_ALIGN(2)
    INPUT_SECTIONS( $OBJECTS(bank_a1))
  } >BANK_A1
  bank_a2 {
    INPUT_SECTION_ALIGN(2)
    INPUT_SECTIONS( $OBJECTS(bank_a2))
  } >BANK_A2
}
```

列表 20：LDF 存储器块分配

```
section("bank_a1") short a[100];
section("bank_a2") short b[100];
```

列表 21：C 源代码中的块分配

C 源代码通过 section("section_name")结构在缓冲声明之前定义区。

需要说明的是，只有全局数据才能直接放入区。更多详细资料见“Blackfin 处理器 VisualDSP++ 2.0 链接器&功能手册”。

附录 A：优化器如何工作

下面用分数标量积循环来说明优化器如何工作。

```
#include <fract.h>
fract32 sp(fract16 *a, fract16 *b) {
  int i;
  fract32 sum=0;
  for (i=0; i<100; i++) {
    sum = add_fr1x32(sum,
    mult_fr1x32(a[i],b[i]));
  }
  return sum;
}
```

列表 22：分数点乘

在生成代码并进行常规标量优化后，编译器将生成如下形式的循环：

```
P2 = 100;
LSETUP(.P1L3, .P1L4 - 2) LC0 = P2;
.P1L3:
R0 = W[P0++] (X);
R2 = W[P1++] (X);
A0 += R0.L * R2.L;
.P1L4:
R0 = A0.w;
```

列表 23：编译输出

退出循环的条件检测已移到最底部，并重新写入循环计数器，计数值减到零，总和累加到 A0。P0 和 P1 分别控制参数 A 和 B 初始化的指针，它们的值也在每次循环中增加。为利用 32 位存储器访问，优化器展开循环，同时并行完成两次循环。Sum 在 A0 和 A1 中累加，且在循环完后累加在一起，并生成最终输出结果。用字加载方式，编译器就必须知道 P0 和 P1 中的初始值长度为四个字节的倍数。还应注意，除非编译器知道初始循环加载了偶数次，否则，就必须在循环外插入奇数次循环的执行条件。

```
P2 = 50;
A1 = A0 = 0;
LSETUP(.P1L3, .P1L4 - 4) LC0 = P2;
.P1L3:
R0 = [P0++];
R2 = [P1++];
A1+=R0.H*R2.H, A0+=R0.L*R2.L;
.P1L4:
R0 = (A0+=A1);
```

列表 24：附加的奇数次循环次数

最后，优化器旋转循环，展开并重新载入循环次数，以得到对各使用功能单元的最高效率。最终生成代码如下：

```
A1=A0=0 || R0 = [P0++] || NOP;
R2 = [I1++];
P2 = 49;
LSETUP(.P1L3,.P1L4-8) LC0 = P2;
.P1L3:
A1+=R0.H*R2.H, A0+=R0.L*R2.L || R0 =
[P0++] || R2 = [I1++];
.P1L4:
A1+=R0.H*R2.H, A0+=R0.L*R2.L;
R0 = (A0+=A1);
```

列表 25：优化器输出

附录 B：编译器转换

编译器支持的优化选项有：

-O	速度优化
-Os	大小优化
-Ox	假定short中的值在16位范围内
-Ofp	改变帧指针的偏移量，使能更短指令的应用
-ipa	执行内部过程优化

表3：优化器相关的命令行选项

关于这些选项的更多详细资料见 VisualDSP++ 2.0 C/C++编译器和 Blackfin 处理器库手册。

参考文献

[1] VisualDSP++ 2.0 C/C++ Compiler and Library Manual for Blackfin processors. First Edition, June 2001. Analog Devices, Inc.

文档记录

Version	Description
May 26, 2003	Updated according to new Blackfin naming conventions and reformatting
December 2001	Initial Release addressing VisualDSP++ version 2.0