ANALOG DEVICES
Technical Notes on using Analog Devices' DSP components and development tools
Contact our technical support by phone: (800) ANALOG-D or e-mail: dsp.support@analog.com
Or visit our on-line resources http://www.analog.com/dsp and http://www.analog.com/dsp/EZAnswers

Tuning C Source Code for the Blackfin® Processor Compiler

Contributed by DSP Tools Compiler Group

May 26, 2003

Introduction

This document provides some guidelines for obtaining the best code execution performance from the Blackfin® processor family's C/C++ compiler using VisualDSP++™ release 2.0.

Use the optimizer

There is a vast difference in the performance of C code that has been compiled optimized and non-optimized. In some cases optimized code can run ten or twenty times faster. Optimization should always be attempted before measuring performance or shipping code as product. Note that the default setting is for non-optimized compilation, the non-optimized default being there to assist programmers in diagnosing problems with their initial coding.

The optimizer in the Blackfin processor compiler is designed to generate efficiently-executing code from C which has been written in a straightforward manner. The basic strategy for tuning a program is to present the algorithm in a way that gives the optimizer excellent visibility of the operations and data, hence the greatest freedom to safely manipulate the code. Note that future releases will enhance the optimizer, and expressing algorithms simply will provide the best path for reaping the benefits of such enhancements.

Use the Statistical Profiler

Tuning source begins with an understanding of what areas of the application are the hot spots. Statistical profiling provided in VisualDSP++ is an excellent means for finding those hot spots.

If the application is unfamiliar to you, compile it with diagnostics and run it unoptimized. This will give you results that connect directly to the C source. You will obtain a more accurate view of your application if you build a fully optimized application and obtain statistics that relate directly to the assembly code. The only problem may be in relating assembly lines to the original source. Do not strip out function names when linking. If you have the function names then you can scroll the assembly window to locate the hot spots. In very complicated code you can locate the exact source lines by counting the loops – unless they are unrolled. Look at the line numbers in the .s file. Note that the compiler optimizer may have moved code around.

Data Types

`char`	8-bit signed integer
`unsigned char`	8-bit unsigned integer
`short`	16-bit signed integer
`unsigned short`	16-bit unsigned integer
`int`	32-bit signed integer
`unsigned int`	32-bit unsigned integer
`long`	32-bit signed integer
`unsigned long`	32-bit unsigned integer

Table 1: Fixed-Point Data Types (Native Arithmetic)

Copyright 2003, Analog Devices, Inc. All rights reserved. Analog Devices assumes no responsibility for customer product design or the use or application of customers' products or for any infringements of patents or rights of others which may result from Analog Devices assistance. All trademarks and logos are property of their respective holders. Information furnished by Analog Devices Applications and Development Tools Engineers is believed to be accurate and reliable, however no responsibility is assumed by Analog Devices regarding technical accuracy and topicality of the content provided in Analog Devices' Engineer-to-Engineer Notes.

The compiler directly supports ten scalar data types as shown in Table 1 and Table 2. double is equivalent to float on Blackfin processors, since 64-bit values are not supported directly on the hardware.

`float`	32-bit floating point
`double`	32-bit floating point

Table 2: Floating-Point Data Types (Emulated Arithmetic)

Fractional data types can be represented as either short or int. Manipulation of these types is best done by using intrinsics, which will be described in a subsequent section.

Avoid Float/Double Arithmetic

Floating-point arithmetic operations are implemented by library routines and, consequently, are far slower than integer operations. An arithmetic floating-point operation inside a loop will prevent the optimizer from using a hardware loop.

Avoid Integer Division in Loops

The hardware does not provide direct support for 32-bit integer division, so the division and modulus operations on int variables are multi-cycle operations. The compiler will convert an integer division by a power of two to a right-shift operation if the value of the divisor is known.

If the compiler has to issue a full division operation, it will issue a call to a library function. In addition to being a multi-cycle operation, this will prevent the optimizer from using a hardware loop for any loops around the division. Whenever possible, do not use divide or modulus operators inside a loop.

Indexed Arrays versus Pointers

C allows you to program data accesses from an array in two ways: either by indexing from an invariant base pointer or by incrementing a pointer. These two versions of the vector addition illustrate the two styles:

```
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];
}
```

Listing 1: Indexed Arrays

```
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++;
}
```

Listing 2: Pointers

Common thought might indicate that the chosen style should not make any difference to the generated code, but sometimes it does. Often, one version of an algorithm will generate better optimized code than the other, but it is not always the same style that is better; the generated code is affected by the surrounding code, which is why there may be differences. The pointer style introduces additional variables that compete with the surrounding code for resources during the optimizer's analysis. Array accesses, on the other hand, must be transformed to pointers by the compiler, and sometimes it does not do the job as well as you could do by hand.

The best strategy is to start with array notation. If this looks unsatisfactory try using pointers. Outside the important loops, use the indexed style, because it is easier to understand.

Use the -ipa Switch

To ensure the best performance, the optimizer often needs to know things that can only be determined by looking outside the routine which it is working on. In particular, it helps to know the alignment and value of pointer parameters and the value of loop bounds. The -ipa compiler switch enables inter-procedural analysis (IPA), which makes this information available. This may be switched on from the IDDE by checking the Interprocedural Optimization box in the Compile tab of the Project Options dialogue selected from the Project menu.

When this switch is used the compiler may be called again from the link phase to recompile the program using additional information obtained during previous compilations.

Because it only operates at link time, the effects of -ipa will not be seen if you compile with the -S switch. To see the assembler file put -save-temps in the Additional Options text box in the Compile tab of the Project Options dialogue and look at the .s file produced after your program has been built.

Much of the following advice assumes that the -ipa switch is being used.

Initialize Constants Statically

Inter-procedural analysis will also identify variables that only have one value and replace them with constants, which can enable better optimization. For this to happen, a variable must have a single value throughout the program.

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

Listing 3: Optimal (IPA knows val is 3)

If the variable is statically initialized to zero, as all global variables are by default, and is subsequently assigned to some other value at another point in the program, then the analysis sees two values and will not consider the variable to have a constant value.

```
#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();
}
```

Listing 4: Non-optimal (IPA cannot see that val is a constant)

Word-align Your Data

To make most efficient use of the hardware, it must be kept fed with data. In many algorithms, the balance of data accesses to computations is such that, to keep the hardware fully utilized, data must be fetched with 32-bit loads.

Although the Blackfin architecture supports byte addressing, the hardware requires that references to memory be naturally aligned. Thus, 16-bit references must be at even address locations, and 32-bit at word-aligned addresses. So, for the most efficient code to be generated, you should ensure that data are word-aligned.

The compiler helps establish the alignment of array data. The stack frames are kept word-aligned. Top-level arrays are allocated at word-aligned addresses, regardless of their data types.

If you write programs that only pass the address of the first element of an array as a parameter and loops that process input arrays an element at a time, starting at element zero, then inter-procedural analysis should be able to establish that the alignment is suitable for 32-bit accesses.

Where the inner loop processes a single row of a multi-dimensional array, be certain that each row begins on a word boundary, possibly inserting dummy data to do so.

Loop Guidelines

Appendix A gives an overview of how the optimizer transforms a loop to generate highly efficient code. It describes the "loop unrolling" optimization technique.

Do not unroll loops yourself

Not only does loop unrolling make the program harder to read but it also prevents optimization. The compiler must be able to unroll the loop itself in order to use wide loads and both accumulators automatically.

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

Listing 5: Optimal (compiler unrolls and uses both computational blocks)

```
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;
  }
}
```

Listing 6: Non-optimal (compiler leaves 16-bit loads)

In this example, the first version of the loop runs almost three times faster than the second, in cases where inter-procedural analysis can determine that the initial values of `a`, `b`, and `c` are aligned on 32-bit boundaries and `n` is a multiple of two.

Avoid loop-carried dependencies

A loop-carried dependency is where computations in a given iteration of a loop cannot be completed without knowing the values calculated in earlier iterations. When a loop has such dependencies, the compiler cannot overlap loop iterations.

Some dependencies are caused by scalar variables that are used before they are defined in a single iteration.

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

Listing 7: Non-optimal (scalar dependency)

An optimizer can reorder iterations in the presence of the class of scalar dependencies known as reductions. These are loops that reduce a vector of values to a scalar value using an associative and commutative operator. The most common example is multiply and accumulate.

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

Listing 8:Optimal (a reduction)

In the first case, the scalar dependency is the subtraction operation; the variable `x` changes in a manner which will give different results if the iterations are performed out of order. In contrast, in the second case, the properties of the addition operator mean that the compiler can perform the operations in any order and still get the same result.

Do not rotate loops by hand

Loops in DSP code are often "rotated" by hand, attempting to do loads and stores from earlier and future iterations at the same time as computation from the current iteration. This technique introduces loop-carried dependencies that prevent the compiler from rearranging the code effectively. It is better to give the compiler a "normalized" version, and leave the rotating to the compiler.

```
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;
}
```

Listing 9: Non-optimal (rotated by hand)

By rotating the loop, the scalar variables `ta` and `tb` have been added, and they have introduced loop-carried dependencies which prevent the compiler from issuing iterations in parallel. The optimizer is capable of doing this kind of loop rotation itself.

```
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;
}
```

Listing 10:Optimal (rotated by the compiler)

Avoid array writes in loops

Other dependencies can be caused by writes to array elements. In the following loop, the optimizer cannot determine whether the load from `a` reads a value defined on a previous iteration or one that will be overwritten in a subsequent iteration.

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

Listing 11: Non-optimal (array dependency)

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

Listing 12: Optimal (induction variables)

The optimizer can resolve access patterns where the addresses are expressions that vary by a fixed amount on each iteration. These are known as "induction variables".

Avoid aliases

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

Listing 13: Non-optimal (potential aliasing)

It may seem that a loop that looks like this does not contain any loop-carried dependencies, but `a` and `b` are both parameters, and, although they are declared with `[ ]`, they are in fact pointers, which may point to the same array. When the same data may be reachable through two pointers, we say they may alias each other.

If the -ipa switch is enabled, the compiler will be able to look at the call sites of `fn` and possibly determine whether they ever point to the same array.

Even with the -ipa switch it is quite easy to create apparent aliases. The interprocedural analysis works by associating pointers with sets of variables that they may refer to at some point in the program. To simplify the analysis no account is taken of the control flow, and if the sets for two pointers are found to intersect then both pointers are assumed to point to the union of the two sets.

If the function `fn` above were to be called in two places with global arrays as arguments, then for the following cases the inter-procedural analysis will have the results shown:

Case 1:

`fn(glob1, glob2, N);`	Sets do not intersect: a and b
`fn(glob1, glob2, N);`	are not aliases (optimal)

Case 2:

`fn(glob1, glob2, N);`	Sets do not intersect: a and b
`fn(glob3, glob4, N);`	are not aliases (optimal)

Case 3:

```
fn(glob1, glob2, N); Sets intersect: a and b
fn(glob3, glob1, N); may be aliases (non-optimal)
```

The third case shows that IPA considers the union of all calls, at once, rather than considering each call individually, when determining whether there is a risk of aliasing. If each call were considered individually, IPA would have to take flow control into account, and the number of permutations would make compilation time impracticably long.

Do as much work as possible in the inner loop

The optimizer focuses on the inner loops because this is where most programs spend the majority of their time. It is considered a good trade-off for an optimization to slow down the code before and after a loop if it is going to make the loop body run faster. So, make sure that your algorithm also spends most of its time in the inner loop. Otherwise, it may actually be made to run slower by optimization.

A useful technique is "loop switching". If you have nested loops where the outer loop runs many times and the inner loop runs a small number of times, it may be possible to rewrite the loops so that the outer loop has fewer iterations.

Avoid conditional code in loops

If a loop contains conditional code, there may be a large penalty incurred if the decision often has to branch against the compiler's prediction. In some cases, the compiler will be able to convert `if-else` and `?:` constructs into conditional moves. In other cases, it will be able to relocate expression evaluation outside of the loop entirely. However, for important loops, linear code should be written.

Keep loops short

For maximum compiler efficiency, loops should be as small as possible. Large loop bodies are usually more complex and difficult to optimize. Additionally, they may require that register data be stored in memory. This will cause a decrease in code density and execution performance.

Do not place function calls in loops

The compiler will not generate hardware loops if the loop contains a function call because of the expense of saving and restoring the context of a hardware loop. In addition to obvious function calls, such as printf(), hardware loop generation will also be prevented by operations such as: integer division and modulus, floating-point arithmetic, and conversion between integer and floating-point data. These operations may require implicit calls to support routines.

Use integers for loop control variables and array indices

For loop control variables and array indices, it is always better to use ints rather than shorts. The C standard states that shorts should be widened to integer sizes before carrying out computation, and then truncated back to short size afterwards.

Frequently, the compiler is able to deal with short loop counters and still detect zero-overhead loops and pointer induction variables. However, it does make the compiler's life harder and may occasionally result in less-optimized code.

Loop pragmas and aiding vectorization

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

Listing 14: Non-optimal (without pragma)

If we call the function `copy` in Listing 14 twice, say `copy(x, y)` and, later, `copy(y, z)`, then interprocedural analysis will not be able to tell that `a` never aliases `b`, as described above. Therefore, the loop contains a loop-carried dependence and cannot be vectorized. A solution in this case is to use the `vector_for` pragma. This tells the compiler that the computation in one iteration of the loop is not dependent on data computed in the previous iteration.

The following code uses the `vector_for` pragma to allow the loop to perform two iterations in parallel.

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

Listing 15: Optimal (with pragma)

Note that this pragma does not force the compiler to vectorize the loop; the optimizer will check various properties of the loop and will not vectorize it if it believes it is unsafe or if it cannot deduce various properties necessary for the vectorization transformation. The pragma assures the compiler that there are no loop-carried dependencies, but there may be other properties of the loop preventing vectorization.

In cases where vectorization is impossible (for example, if array `a` were aligned on a word boundary, but `b` were not), then the information given in the assertion made by `vector_for` may still be put to good use in aiding other optimizations.

Const Data is Constant

By default, the compiler will assume that the data pointed to by a pointer to `const` data will not change. Therefore, another way to tell the compiler that the two arrays `a` and `b` do not overlap is to use the `const` keyword.

The example in Listing 16 will have a similar effect to the `vector_for` pragma. In fact, the `const` implementation is better since it also allows the optimizer, after vectorization, to rotate the loop, which requires knowing that it is not just adjacent iterations of the loop which have no dependence on each other, but iterations further apart, too.

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

Listing 16: Usage of const keyword

In C, it is legal, though bad programming practice, to use casts to allow the data pointed at by pointers to `const` data to change. This should be avoided since, by default, the compiler will generate code that assumes `const` data does not change. However, if you have a program that modifies `const` data through a pointer, you can generate correct code by using the compile-time flag -const-read-write.

Fractional Data

Fractional data, represented as 16-bit and 32-bit integers, can be manipulated in two ways. The recommended way, giving you the most control over your data, is by use of intrinsics. Let us consider the fractional scalar product. This may be written as:

```
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;
}
```

Listing 17: Non-optimal (uses shifts)

However, this presents some problems to the optimizer. Normally, the code generated here would be a multiply, followed by a shift, followed by an accumulation. However, the Blackfin processor has a fractional multiply accumulate instruction that performs all these tasks in one cycle. Moreover, it can do two of these instructions in parallel.

The compiler recognizes this idiom and acknowledges that, in the DSP world, the preference is for saturating arithmetic. The multiply/shift is replaced by a saturating fractional multiply. The transformation can be disabled by using the -no_int_to_fract switch in case saturation is not required.

However, this was a simple case. In more complicated cases, where perhaps the multiply is further separated from the shift, the compiler may not detect the possibility of using a fractional multiply. The recommended coding style is to use the intrinsics. In the following example, `add_fr1x32()` and `mult_fr1x32` are used to add and multiply fractional 32-bit data, respectively.

```
#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;
}
```

Listing 18: Optimal (uses intrinsics)

The full list of fractional operations is given in the Blackfin processor C/C++ Compiler manual, together with descriptions of other intrinsics available. The intrinsic functions provide operations that generally operate on single 16- or 32-bit values; the compiler will recognize when a loop can be vectorized and will generate 2x16 operations in such circumstances. Just as it is better to leave loop rotation to the compiler, the intrinsics leave operation-pairing to the compiler as well.

If Possible put Arrays into Different Memory Sections

The Blackfin processor can support two memory operations on a single instruction line. However, this will only complete in one cycle if the two addresses are situated in different memory blocks; if both access the same block, then a stall will be incurred. Take as an example the dot product (as shown in the previous section).

Because data is loaded from arrays `a` and `b` in every cycle, it may be useful to ensure that these arrays are located in different memory blocks. As an example, consider defining two banks in the MEMORY portion of the .LDF file.

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

Listing 19: LDF Memory Layout

Then, configure the SECTIONS portion to tell the linker to place data sections in specific memory banks, as shown below.

```
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
}
```

Listing 20: LDF Section Assignment

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

Listing 21: Section Assignment in C sources

In the C source code, sections are defined with the `section("section_name")` construct preceding a buffer declaration.

Note that explicit placement of data in sections can only be done for global data. Please see the "VisualDSP++ 2.0 Linker & Utilities Manual for Blackfin processors" for further details.

Appendix A: How the optimizer works

We will use the following fractional scalar product loop to show how the optimizer works.

```
#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;
}
```

Listing 22: Fractional Dot Product

After code generation and conventional scalar optimizations, the compiler will have generated a loop that looks something like this:

```
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;
```

Listing 23: Compiler output

The loop exit test has been moved to the bottom and the loop counter rewritten to count down to zero. The sum is being accumulated in A0. P0 and P1 hold pointers that are initialized with the parameters `A` and `B`, respectively, and are incremented on each iteration. In order to use 32-bit memory accesses, the optimizer unrolls the loop to run two iterations in parallel. `sum` is now being accumulated in A0 and A1, which must be added together after the loop to produce the final result. In order to use word loads, the compiler has to know that P0 and P1 have initial values that are multiples of four bytes. Note also that, unless the compiler knows that original loop was executed an even number of times, a conditionally-executed odd iteration must be inserted outside the loop.

```
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);
```

Listing 24: Additional Odd Iteration

Finally the optimizer rotates the loop, unrolling and overlapping iterations to obtain highest possible use of functional units. The following code is finally generated:

```
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);
```

Listing 25: Optimizer output

Appendix B: Compiler switches

The optimization switches supported by the compiler are:

-O	optimize for speed
-Os	optimize for size
-Ox	assume values in `short`s remain in 16-bit range
-Ofp	change frame pointer offsets to enable use of shorter instructions
-ipa	do inter-procedural optimization

Table 3: Optimizer-related command-line switches

More details of these switches can be found in the .VisualDSP++ 2.0 C/C++ Compiler and Library Manual for Blackfin processors..

References

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

Document History

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