

Ever Faster Float Comparisons

Randy Dillon

I was reading the latest Game Programming Gems (#6), and it has a gem on floating point tricks [Lomont06], a topic in which I've been interested in for a while. Specifically, several months ago I started working on comparing floats as integers (one of the topics of the gem) in order to avoid the native floating point compare instruction which on some platforms can be painfully slow. The gem led me back to reviewing [Lomont05] and [Dawson05] on the subject, and at the same time as I was learning from their discussions, I became aware of room for improvement in the recommended compare function presented by [Lomont05,06]. After doing some googling I could not find any articles that mentioned the same modifications I've applied, so here they are for your reading/coding pleasure along with some other investigations.

First I'll mention I am not going to fully review the IEEE 754 floating point format yet again, [Lomont05,06] and [Dawson05] do an excellent job of that already if you need a refresher. Also, for an explanation of LomontCompare and its evolution please refer to [Lomont05]. I'll *briefly* cover just enough of IEEE 754 to suffice for the purposes of this article.

Brief Review of IEEE 754

IEEE 754 floats are stored in a sign-magnitude format (as opposed to two's complement format, used for integers in most modern CPUs). For 32 bit floats (there are other sizes of IEEE 754 floats) Bit 31 is the sign bit, and bits 0..30 are the unsigned magnitude of the float. Although the magnitude actually has a two-part format, *for our purposes of comparison* we can simply treat the magnitude as an integer value. The sign-magnitude format gives rise to two version of zero: -0 is 0x80000000, with magnitude 0 and sign bit 1; +0 is 0x00000000, magnitude zero and sign bit 0. IEEE 754 says these two versions of zero are mathematically identical (i.e. $-0 = +0 = 0$). 0x00000001 and 0x80000001 are respectively the smallest (closest to zero) positive and negative non-zero float values (as opposed to 0x00000001 and 0xffffffff for two's complement integers). For all the other details, please see one of the referenced papers.

LomontCompare

So, down to the meat of it. Lets start by reviewing the code for LomontCompare. Here is a version comprising a combination of versions from [Lomont05] and [Lomont06], where other than retaining the use of (included) macro SIGNMASK, I've opted for a version that uses no other externally defined macros/functions so we can see everything that is happening...

```

#ifdef _SIGNED_SHIFT
#define SIGNMASK(i) ((int)(i)>>31)
#else
#define SIGNMASK(i) ( (int)(~( ((unsigned int)(i))>>31)-1) )
#endif

bool LomontCompare(float af, float bf, int maxDiff)
{ // solid, fast routine across all platforms
  // with constant time behavior
  int ai = *reinterpret_cast<int*>(&af);
  int bi = *reinterpret_cast<int*>(&bf);
  int test = SIGNMASK(ai^bi);
  assert((0 == test) || (0xFFFFFFFF == test));
  int diff = ((0x80000000 - ai)&(test)) | (ai & ~test) - bi;
  int v1 = maxDiff + diff;
  int v2 = maxDiff - diff;
  return (v1|v2) >= 0;
}

```

It is hard to *greatly* improve on what is already presented. The improvements Lomont offered to DawsonCompare, after asking “Can we make it faster yet?” centered chiefly around elimination of branching, and the result is a huge win, especially for heavily pipelined architectures, providing a fast constant time routine with little room for improvement.

Still, we’ll ask the same question about LomontCompare as Lomont asked about DawsonCompare: Can we make it faster yet? What could we possibly improve?

Faster Portable SIGNMASK

Well, for starters if you’ve read [Lomont05] you may notice that in LomontCompare3 (the recommended version in that paper) the mask <test> uses reversed semantics as compared to the version shown here using SIGNMASK. LomontCompare3 calculates <test> so that it is 0 when <ai^bi> indicates differing signs, and 0xffffffff when they are the same.

```
test = ((unsigned int)(ai^b1) >> 31) - 1;
```

This uses one less operation than the portable version of SIGNMASK, omitting the final bitwise NOT (~). Of course, when using the SIGNMASK macro we want to settle on a single consistent semantic, and the one chosen by [Lomont06] is the one that matches the non-portable, fastest version, which works only if signed right shifts preserve the sign bit.

```
#define SIGNMASK(i) ((int)(i)>>31)
```

To match this semantically, the portable version applies the extra bitwise NOT (~) to invert the masking semantic.

```
#define SIGNMASK(i) ((int)(~(((unsigned int)(i))>>31)-1))
```

We can also say, it shifts the sign bit down to bit 0, leaving a 32 bit value of 0 or 1, and then takes the two's complement via subtracting 1 and then taking the one's complement. (Alternatively, the two's complement can be effected by first taking the one's complement and then adding 1). So, here is the first place we'll make a change, by speeding up the portable version of SIGNMASK. Taking the two's complement is a fancy way of saying "negate", so we'll tell the compiler do that for us directly which reduces the number of operations by 1.

```
#define SIGNMASK(i)  (~(int)(((unsigned int)(i))>>31))
```

A compiler might *possibly* (*i.e. not likely*) notice the previous version is doing a manual two's complement and replace it with a direct "neg" instruction (By the way, MSVC++ Express Edition does *not* make this optimization, while it *does* sometimes surprise me with other optimizations it performs), but there is no point in expressing what we want with more operations than necessary and hoping for a compiler to be smart enough to know what we are doing at this level. Now we've ensured the portable version uses the minimum number of operations while retaining the same masking semantic.

Faster conditional use of "reordered" <ai>

Our next target is going to be the conditional use of the "reordered" version of <ai> vs the original <ai>.

Before attempting to change something it is good to understand it. Lets examine the patient before we wheel it into surgery.

```
((0x80000000 - ai)&(~test)) | (ai & test))
```

Simply put, this calculates the "reordered to two's complement" version of <ai> (see [Dawson05]), and then uses <test> and it's inverse as a way to "mask out" one version of ai and "mask in" the other. We'd like a faster way of making this selection. The current method uses 4 bitwise operations to effect the selection. There is a way to do this in 3 operations (no pun intended), by making use of a property of the XOR (Exclusive OR) operation. Recall that XORing a value X with a value Y and then XORing the result with Y again effectively cancels the first XOR with Y, leaving the original value X. So, given two values, X & Y and our mask <test>, we can do this...

```
(((X ^ Y) & test) ^ Y)
```

Now, when test is 0, the X^Y when ANDed against 0 gives 0, leaving us with 0^Y which of course is simply Y. When test is 0xffffffff, then the AND gives us the result X^Y, and applying the final ^Y gives X^Y^Y, and since the two Ys cancel out, we are left with X.

So, given this formula the rule is: The final (rightmost) value to be XORed (twice overall) is the one that will be selected when <test> is 0, and the other will be selected when <test> is 0xffffffff. In our case, <test> is 0xffffffff when we want (0x80000000 - ai) as the result. Substitute (0x80000000 - ai) for X and <ai> for Y ...

```
(((0x80000000 - ai) ^ ai) & test) ^ ai)
```

Incidentally, it is worth noting that there is an alternate form using subtraction and addition instead of XOR, producing the same result (similar to how two ints can be swapped in-place with 3 XORs, or with an add and two subtracts).

```
(((0x80000000 - ai) - ai) & test) + ai)
```

So, we end up with the following version. I've inserted my initials into the function name as an indication of my modification...

```
#ifdef _SIGNED_SHIFT
#define SIGNMASK(i) ((i)>>31)
#else
#define SIGNMASK(i) (-(int)(((unsigned int)(i))>>31))
#endif

bool LomontRRDCompare1(float af, float bf, int maxDiff)
{ // solid, fast routine across all platforms
  // with constant time behavior
  int ai = *reinterpret_cast<int*>(&af);
  int bi = *reinterpret_cast<int*>(&bf);
  int test = SIGNMASK(ai^bi);
  assert((0 == test) || (0xFFFFFFFF == test));
  int diff = (((0x80000000 - ai) ^ ai) & test) ^ ai) - bi;
  int v1 = maxDiff + diff;
  int v2 = maxDiff - diff;
  return (v1|v2) >= 0;
}
```

We've now reduced the number of operations by 2 for the portable version, and by 1 for the non-portable version. Could we do better?

Recall from [Dawson05] that for negative sign-magnitude numbers, in order to be able to compare them in the same way with both positive and negative numbers we “adjust them so that they are lexicographically ordered as twos-complement integers instead of as sign-magnitude integers. This is done by detecting negative numbers and subtracting them from 0x80000000.” We can think of this as transforming the number into a new numerical space. For positive sign-magnitude numbers, the numerical space they are in is already equivalent to two's complement numerical space. (Or we could say the positive half of the sign-magnitude space maps directly to the positive half of the two's complement space). With two numbers in the same numerical space, we can meaningfully compare any combination of negative and positive. In *this* function however, we adjust the value of <ai> not based on it being negative, but based on its sign being different from <bi> (i.e., they are in different numerical spaces), so sometimes we can be transforming <ai> when it is positive, and we could say that in that case we transform it into its equivalent value in the negative half of sign-magnitude space (which <bi> is in in this case), even though <ai> is positive. Kind of strange to wrap one's head around, but that is how I conceptualize it. The bottom line is, subtracting <ai> from

0x80000000 effectively negates (takes the two's complement of) the magnitude bits, and leaves the sign bit unchanged (except for -0 0x80000000, becoming +0 0x00000000).

Recall from the previous SIGNMASK discussion that another way to take the two's complement of a number is to take the one's complement (i.e. bitwise NOT (~)), and then add 1. As you may know, especially if you have done a lot of assembly programming, given a control <mask> with all bits set when we want to take the two's complement of an integer <value> and all bits clear when we want the unchanged <value>, the two's complement can be conditionally calculated as follows.

$$(\text{value} \wedge \text{mask}) - \text{mask}$$

If <mask> is 0, well obviously there is no change. If it is 0xffffffff, the xor takes the one's complement, and then subtracting it (0xffffffff == -1) effectively adds 1, completing the two's complement. Now, this takes two operations to calculate the two's complement vs the one operation in (0x80000000 - <value>), but it directly involves <mask> in the calculation of the two's complement, so no additional conditional "selection" code using <mask> is needed. We're not quite done yet though. Recall (0x80000000 - value) takes the two's complement of the magnitude bits only, leaving the sign bit unmodified. So, we require the modification...

$$(\text{value} \wedge (\text{mask} \& 0x7fffffff)) - \text{mask}$$

Giving us this version of the function.

```
bool LomontRRDCompare2(float af, float bf, int maxDiff)
{ // solid, fast routine across all platforms
  // with constant time behavior
  int ai = *reinterpret_cast<int*>(&af);
  int bi = *reinterpret_cast<int*>(&bf);
  int test = SIGNMASK(ai^bi);
  assert((0 == test) || (0xFFFFFFFF == test));
  int diff = ((ai ^ (test & 0x7fffffff)) - test) - bi;
  int v1 = maxDiff + diff;
  int v2 = maxDiff - diff;
  return (v1|v2) >= 0;
}
```

So, this is a reduction from the original LomontCompare of 3 operations for the portable version and 2 for the non-portable version. Further, as used in our expression with the AND against 0x7fffffff, re-ordering the two's complement operations (i.e. first subtract 1, then take ones complement) results in a slightly more pipeline friendly expression in general, and with MSVC++ in particular eliminates the need for the generated code to move the unmodified <test> to a temporary register for later use prior to executing (test & 0x7fffffff), and also eliminates the need to save and restore a register...

```

bool LomontRRDCompare3(float af, float bf, int maxDiff)
{ // solid, fast routine across all platforms
  // with constant time behavior
  int ai = *reinterpret_cast<int*>(&af);
  int bi = *reinterpret_cast<int*>(&bf);
  int test = SIGNMASK(ai^bi);
  assert((0 == test) || (0xFFFFFFFF == test));
  int diff = ((ai + test) ^ (test & 0x7fffffff)) - bi;
  int v1 = maxDiff + diff;
  int v2 = maxDiff - diff;
  return (v1|v2) >= 0;
}

```

This gives us a fairly optimal implementation giving us exactly the same results as the original LomontCompare. Now, there is *another* thing we can do to make it faster, but here the results diverge slightly from the original. We can forego completing the two's complement, removing the addition of <test> which effects the -1 , leaving us instead using the one's complement.

```

bool LomontRRDCompare4(float af, float bf, int maxDiff)
{ // solid, fast routine across all platforms
  // with constant time behavior
  int ai = *reinterpret_cast<int*>(&af);
  int bi = *reinterpret_cast<int*>(&bf);
  int test = SIGNMASK(ai^bi);
  assert((0 == test) || (0xFFFFFFFF == test));
  int diff = (ai ^ (test & 0x7fffffff)) - bi;
  int v1 = maxDiff + diff;
  int v2 = maxDiff - diff;
  return (v1|v2) >= 0;
}

```

So what is the effect of using the one's complement? Float -0 becomes integer -1 instead of 0 , the smallest (i.e. closest to zero) non-zero negative float becomes integer -2 instead of -1 , etc. - the output result is off by -1 . When <ai> and <bi> are opposite signs this effectively reduces or increases the tolerance value by 1 for ai or bi respectively being negative, and when <ai> and <bi> are the same sign the tolerance is effectively unchanged. For float values of order 1 magnitude (e.g. 10), this results in a difference in the applied tolerance of (if I am counting my bits correctly) $\pm (\frac{1}{2}^{23})/10 \approx \pm 1/838,860$ or roughly ± 0.0000012 (Remember, the tolerance scales with the lower magnitude input). Actually, this is the upper range. See [Dawson05], 2nd to last paragraph of section "Comparing using integers" for details on how the tolerance value is not actually linearly centered.

Since we are dealing with a tolerance value anyway, and a non-centered one to boot, it behooves us to consider how important the accuracy of the tolerance is as compared to optimal speed (although the speed difference is not large). Only you can answer this question for the particular application you would use a function like this for.

The difference in performance between these various versions will vary depending on the target architecture and the compiler. Here is a breakdown of the number of assembly

instructions, as well as test times relative to the original LomontCompare, for the preceding versions of this compare function on my PC compiled with MSVC++ 2005 Express Edition, optimized for speed, not inlined, without Link Time Code Generation. With LTCG, and similarly for inlining, the generated code depends much more on the various contexts it is called from (i.e. the test code itself impacts the code generated for the compare functions). Without LTCG or inlining we get a much more “stand-alone” version of each implementation. Float values get passed as parameters on the call stack, <maxDiff> is passed in a register.

	<u>Instructions</u>	<u>Relative Time</u>
LomontCompare	24	100%
LomontCompareb	23	95%
LomontRRDCompare1	22	93.5%
LomontRRDCompare2	21	93%
LomontRRDCompare3	18	86.5%
LomontRRDCompare4	17	84.5%

LomontCompareb is LomontCompare using the updated version of SIGNMASK. The timings do not include test code overhead. A dummy function which does essentially nothing except incur the function call overhead and return an unchanging Boolean value is called from a duplicate of the timing code used for the actual compare functions, and the test times for the dummy function are subtracted from the test times for the compares, so the presented relative times will pretty closely reflect the actual performance difference of the various function implementations.

My Homework

One of the things Chris Lomont leaves us with in [Lomont05] is some homework items, one of which is:

Create similar routines for comparing floats using greater or less than comparisons with a little padding, i.e., a routine with the signature

```
bool FloatLess(float a, float b, int padding);
```

Ok, well my first thought is, lets start with LomontRRDCompare and modify it to do the “Less than with padding test”. Hmm, what exactly would that test be if we start with the current function? The beauty of the existing compare function is that it does not care which input is bigger or smaller, and this is exactly what allows it to work the way it does. It makes no attempt to determine or keep track of which input is larger or smaller. I toyed around with the idea of keeping track of sign bits and modifying results based on the different ways they are set. But this is going about things bass-ackwards I think. The routine as-is was simply not written with that in mind, we’d just be butchering it.

So, we’ll try a fresh approach for our fresh new problem. Unlike for LomontRRDCompare(), we realize that for FloatLess() we are of course *entirely* concerned with which input is bigger or smaller. One goal will be to once again avoid any branching. The approach we’ll take is to refer back to [Dawson05] and how

DawsonCompare converted negative inputs to their re-ordered counterpart (including converting -0 to 0) as needed so the two values could be directly compared, except of course, we want to do that without branching. Lets dive into it using what we know from LomontRRDCompare. We'll create a control mask for each input , which we'll use to conditionally convert negative values to their properly ordered negative counterparts.

```
bool DillonCompareLess1 (float af, float bf, int padding)
{
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int testa = SIGNMASK(ai);
    int testb = SIGNMASK(bi);
    ai = (ai + testa) ^ (testa & 0x7fffffff);
    bi = (bi + testb) ^ (testb & 0x7fffffff);
    return ai + padding < bi;
}
```

Note that in this case we know the re-ordering will only occur for each input when it is negative, which gives us more options regarding ANDing against $0x7fffffff$ (we can do that against either the input or its mask), however we'll stick with the same pipeline friendly pattern as we used for LomontRRDCompare3. Also note that in the final statement we don't take the difference between $\langle ai \rangle$ and $\langle bi \rangle$, avoiding potential wrap-around issues for all but the very largest magnitude numbers approaching the single precision float limits (How large actually depends on the magnitude of padding).

Once again, we have the same option as we had after arriving at LomontRRDCompare3. We can opt to simplify the conditional two's complement to a conditional one's complement and knock another two operations out of the function, giving us this fairly tight bit of code...

```
bool DillonCompareLess2(float af, float bf, int padding)
{
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int testa = SIGNMASK(ai);
    int testb = SIGNMASK(bi);
    ai = (ai & 0x7fffffff) ^ testa;
    bi = (bi & 0x7fffffff) ^ testb;
    return ai + padding < bi;
}
```

All I need now is for Mr. Lomont to grade my homework ☺

Other Investigations

So, [Lomont06] presents some nice fast direct comparisons against zero. How about a comparison against zero with tolerance? We can simply compare the absolute value of the magnitude against the tolerance, which should be a positive value. Very simple, including a variation which takes a float tolerance.

```

bool CompareZero(float f, int maxDiff)
{
    int fi = *reinterpret_cast<int*>(&f);
    return (fi & 0x7fffffff) <= maxDiff;
}

bool CompareZero(float f, float tolerance)
{
    int fi = *reinterpret_cast<int*>(&f);
    int ti = *reinterpret_cast<int*>(&tolerance);
    return (fi & 0x7fffffff) <= ti;
}

```

As authors of code, we programmers often (hopefully) know the range of values a particular variable can validly hold at any particular time. Specifically, there are times when we know a value will always be positive or always be negative at the time when we want to compare it to another value. An interesting thing to note is that we can do a *very* fast direct compare between two floats when we know that at least one is positive, or when we know that at least one is negative.

```

bool FloatLessPositive(float af, float bf)
{ // return true when a less than b
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    assert((ai&bi) >= 0); // one or both must be positive
    return ai < bi;
}

bool FloatLessNegative(float af, float bf)
{ // return true when a less than b
    unsigned int ai = *reinterpret_cast<unsigned int*>(&af);
    unsigned int bi = *reinterpret_cast<unsigned int*>(&bf);
    assert((int)(ai|bi) < 0); // one or both must be negative
    return ai > bi; // inverted comparison of unsigned int
}

```

The positive case is pretty straight ahead, but what about that negative case? How does the inverted comparison of unsigned int work? Well, the smallest negative float value (-0) is 0x80000000. Interpreted as an unsigned (i.e. positive) integer, this is larger than all values with the sign bit clear (which represent positive float values), so that handles the case of differing signs. What about when both values are negative? The smallest (i.e. closest to 0) non-zero negative float value is 0x80000001, and as an unsigned value is larger than (-0) 0x80000000, which means it is a higher magnitude negative number (or less than -0). So as the value of numbers larger than unsigned 0x80000000 increases, so does their magnitude as negative numbers. The caveat for these functions is they do not properly handle -0, in that where IEEE 754 says -0 == 0, these functions consider -0 less than 0. You need to make sure that if you are considering 0 as one of your positive values, that it is not actually -0, (i.e. you don't want to pass -0 and another negative value to the positive compare function). Even with the caveat, this approach can be very useful.

We can combine these two ideas into one general direct compare function. Worth noting is if the two values are opposite signs, the Positive and Negative path both produce the correct result.

```
bool FloatLess1(float af, float bf)
{
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    if (ai >= 0)    // at least one input is positive
        return ai < bi;
    else           // at least one input is negative.
        return (unsigned int)ai > (unsigned int)bi;
}
```

`FloatLess1` takes about 85% of the time of a native float comparison on my PC when inlined (Which I especially recommend for small functions like this or you can lose more to function call overhead than you could have gained). On newer PCs, it is slower than native. On an XBOX 360 this approach takes only about 50% of the native float compare time when inlined and passing inputs by reference instead of by value to avoid incurring a Load Hit Store penalty (See Platform Considerations section).

Can we somehow apply this logic without the branch and increase our speed over the average case of `FloatLess1`? Well, lets see. Applying the XOR and mask selection method from `LomontRRDCompare1`, we would calculate both results and select the appropriate one.

```
bool FloatLess2(float af, float bf)
{
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    bool negResult = (unsigned int)ai > (unsigned int)bi;
    bool posResult = ai < bi;
    // if ai is negative, we'll take the negative result.
    int test = SIGNMASK(ai);
    return ((negResult ^ posResult) & test) ^ posResult;
}
```

Testing reveals that `FloatLess2` takes about 109% of the time of a native float compare on my PC. We're doing enough work in the constant time version that it outweighs any gain we get by avoiding the branch, even in the worst case of always taking the branch..

To gain anything from a non-branching version we need to simplify. There is another strategy we can try besides calculating both results and then selecting between them. We know from the discussion leading to `FloatLess1` that we can use the standard comparison if the input signs differ or both are positive, and also that we can use either the standard or alternate comparison when the input signs differ. In the case of *both* being negative, we *must* use an alternate comparison – or – from another perspective, knowing both inputs are negative *allows* us to apply the same transformation to both inputs. I.e. we can conditionally invert both inputs based on one control mask, and then always use the standard comparison.

```

bool FloatLess3(float af, float bf)
{
    int ai = *reinterpret_cast<int*>(&af);
    int bi = *reinterpret_cast<int*>(&bf);
    int test = SIGNMASK(ai&bi); // all bits on when both < 0.
    return (ai ^ test) < (bi ^ test);
}

```

Note that we only need to *invert* to be able to use the standard comparison, we don't need to *lexographically re-order*. Incredibly, on my PC, compiled with MSVC++ 2005 Express Edition, the test runs average about 112% of native compare time. Inspecting the generated code reveals the compiler using 15 instructions to do what can be done in 10, or 11 at most, so there would seem to be potential for greater performance depending on your target and/or compiler. On the XBOX 360, this approach (with the non-portable SIGNMASK) is not quite as fast as FloatLess1, coming in at about 55% of native float compare time. Well, this is about the best I can come up with for a non-branching version in C++. Once again, check on your target hardware.

Platform Considerations

When using these techniques, we need to be very careful on some platforms. Specifically, the CPU on the PS3 and Xbox 360 can incur something called a Load-Hit-Store penalty. Load-Hit-Store (LHS) happens when you write to a memory location, and then immediately or very soon afterwards read from that same location. The memory/cache architecture is such that the memory write will not have finished, and you will essentially stall the CPU (and also flush the instruction pipeline). Your attempt to "Load" has "Hit" a recent "Store" to the same location and can incur up to 60 cycles of delay, whereas the fcmp instruction we're avoiding has a 10 cycle latency on this CPU. The only way these machines can move a value from a float register into an integer register is via storing it into memory. Floats passed by value are passed in float registers, so passing the float by value is going to cause these otherwise fast float comparisons (and any other function that loads the raw bit pattern of the float) to immediately store it into memory right before loading it into an int register, incurring an LHS, causing worse performance than using the native float instructions would have in the first place.

What is a poor programmer to do? Well, you can make these kinds of functions take a reference to const float (const float&) for any float parameters. This way, if you are passing a float that is already residing in memory, the address of the float will actually be passed to the function, and the float can be loaded directly from memory into the integer register without necessarily incurring an LHS (This depends how recently the value was written to memory, but at least you are no longer guaranteeing an LHS every time). As for passing floats that are the result of an immediately preceding calculation, sometimes you need to find a way to do some other work between doing the calculation and calling the float function, and sometimes just don't use these techniques, since you *can* actually degrade performance under certain conditions. (Actually, if you make the function parameters const float*, you'll prevent yourself from accidently directly passing the result of a float calculation, since the compiler will refuse to digest it). You also need to

make sure the result gets stored in memory (e.g. the stack) well before calling the function. As the saying goes, with great power comes great responsibility.

Inlining vs not inlining, pass by value vs pass by reference, the compiler used, the target architecture and the context in which you use these techniques all play a part in the performance difference that can be achieved. To maximize the gain, you need to bring an understanding of the relevant issues to the table (which sometimes means examining the generated code and being very surprised) and discern where to apply these techniques or not. Lastly, what does increase performance on one platform may actually degrade performance on another, so as always, run tests on your target(s) and use accordingly.

Closing Comments

I've presented some improvements to an existing technique, and explored some new techniques (new to me at least). While doing so, the code practices shown here were not always the best, e.g. directly using `reinterpret_cast` to access the floats, which is not completely portable. The goal was to make everything visible rather than hidden away in other un-included code. In actual practice, it is best to use accessor functions/macros as [Lomont06] discusses, and use named values rather than magic numbers in code.

Much gratitude to Chris Lomont and Bruce Dawson for their excellent articles.

Hopefully this has provided some useful information and perhaps also some inspiration to do some of your own fast float investigation and come up with further improvements to existing approaches as well as whole new approaches. Please let me know of anything interesting you come up with, and also any errors or oversights on my part. Thanks for reading, now go write some code!

References:

[Lomont05] – Lomont, Chris. “Taming the Floating Point Beast”, 2005,
<http://www.lomont.org/Math/Papers/2005/CompareFloat.pdf>

[Lomont06] – Lomont, Chris. “Floating Point Tricks”: Game Programming Gems #6, 2006, pg 121

[Dawson05] – Dawson, Bruce. “Comparing Floating Point Numbers”, 2005
<http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm>

Wikipedia – Two's Complement:
http://en.wikipedia.org/wiki/Two%27s_complement

Wikipedia – IEEE 754:
http://en.wikipedia.org/wiki/IEEE_754

email: randy@randydillon.org