There Are Only Four Billion Floats–So Test Them All (2014)

Source: randomascii.wordpress.com
75 points by albrewer 6 hours ago on hackernews | 12 comments

A few months ago I saw a blog post touting fancy new SSE3 functions for implementing vector *floor*, *ceil*, and *round* functions. There was the inevitable proud proclaiming of impressive performance and correctness. However the *ceil* function gave the wrong answer for many numbers it was supposed to handle, including odd-ball numbers like 'one'.

The *floor* and *round* functions were similarly flawed. The reddit discussion of these problems then discussed two other sets of vector math functions. Both of them were similarly buggy.

Fixed versions of some of these functions were produced, and they are greatly improved, but some of them still have bugs.

Floating-point math is hard, but testing these functions is trivial, and fast. Just do it.

The functions *ceil, floor,* and *round* are particularly easy to test because there are presumed-good CRT (C RunTime) functions that you can check them against. And, you can test every float bit-pattern (all four billion!) in about ninety seconds. It's actually very easy. Just iterate through all four-billion (technically $2 \wedge 32$) bit patterns, call your test function, call your reference function, and make sure the results match. Properly comparing NaN and zero results takes a bit of care but it's still not too bad.

> Aside: floating-point math has a reputation for producing results that are unpredictably wrong. This reputation is then used to justify sloppiness, which then justifies the reputation. In fact IEEE floating-point math is designed to,

whenever practical, give the best possible answer (correctly rounded), and functions that extend floating-point math should follow this pattern, and only deviate from it when it is clear that correctness is too expensive.

Later on I'll show the implementation for my *ExhaustiveTest* function but for now here is the function declaration:

```
typedef float(*Transform)(float);

// Pass in a range of float representations to compar
// start and stop are inclusive. Pass in 0, 0xFFFFFF
// floats. The floats are iterated through by increme
// their integer representation.
void ExhaustiveTest(uint32_t start, uint32_t stop, Tr
            Transform RefFunc, const char* desc)
```

Typical test code that uses ExhaustiveTest is shown below. In this case I am testing the original SSE 2 *_mm_ceil_ps2* function that started the discussion, with a wrapper to translate between float and __m128. The function didn't claim to handle floats outside of the range of 32-bit integers so I restricted the test range to just those numbers:

```
float old_mm_ceil_ps2(float f)
{
    __m128 input = { f, 0, 0, 0 };
    __m128 result = old_mm_ceil_ps2(input);
    return result.m128_f32[0];
}

int main()
{
    // This is the biggest number that can be represe
    // both float and int32_t. It's 2^31-128.
    Float_t maxfloatasint(2147483520.0f);
    const uint32_t signBit = 0×80000000;
    ExhaustiveTest(0, (uint32_t)maxfloatasint.i, old_
```

```
                "old _mm_ceil_ps2");
    ExhaustiveTest(signBit, signBit | maxfloatasint.i
                "old _mm_ceil_ps2");
}
```

Note that this code uses the Float_t type to get the integer representation of a particular float. I described Float_t years ago in Tricks With the Floating-Point Format .

How did the original functions do?

_mm_ceil_ps2 claimed to handle all numbers in the range of 32-bit integers, which is already ignoring about 38% of floating-point numbers. Even in that limited range it had 872,415,233 errors – that's a 33% failure rate over the 2,650,800,128 floats it tried to handle. *_mm_ceil_ps2* got the wrong answer for all numbers between 0.0 and *FLT_EPSILON* * 0.25, all odd numbers below 8,388,608, and a few other numbers. A fixed version was quickly produced after the errors were pointed out.

Another set of vector math functions that was discussed was DirectXMath. The 3.03 version of DirectXMath's *XMVectorCeiling* claimed to handle all floats. However it failed on lots of tiny numbers, and on most odd numbers. In total there were 880,803,839 errors out of the 4,294,967,296 numbers (all floats) that it tried to handle. The one redeeming point for *XMVectorCeiling* is that these bugs have been known and fixed for a while, but you need the latest Windows SDK (comes with VS 2013) in order to get the fixed 3.06 version. And even the 3.06 version doesn't entirely fix *XMVectorRound*.

The LiraNuna / glsl-sse2 family of functions were the final set of math functions that were mentioned. The LiraNuna *ceil* function claimed to handle all floats but it gave the wrong answer on 864,026,625 numbers. That's better than the others, but not by much.

I didn't exhaustively test the *floor* and *round* functions because it would complicate this article and wouldn't add significant value. Suffice it to say that they have similar errors.

Sources of error

Several of the *ceil* functions were implemented by adding 0.5 to the input value and rounding to nearest. This does not work. This technique fails in several ways:

1. Round to nearest even is the default IEEE rounding mode. This means that 5.5 rounds to 6, and 6.5 also rounds to 6. That's why many of the *ceil* functions fail on odd integers. This technique also fails on the largest float smaller than 1.0 because this plus 0.5 gives 1.5 which rounds to 2.0.
2. For very small numbers (less than about *FLT_EPSILON* * 0.25) adding 0.5 gives 0.5 exactly, and this then rounds to zero. Since about 40% of the positive floating-point numbers are smaller than FLT_EPSILON*0.25 this results in a lot of errors – over 850 million of them!

The 3.03 version of DirectXMath's *XMVectorCeiling* used a variant of this technique. Instead of adding 0.5 they added *g_XMOneHalfMinusEpsilon*. Perversely enough the value of this constant doesn't match its name – it's actually one half minus 0.75 times *FLT_EPSILON*. Curious. Using this constant avoids errors on 1.0f but it still fails on small numbers and on odd numbers greater than one.

NaN handling

The fixed version of *_mm_ceil_ps2* comes with a handy template function that can be used to extend it to support the full range of floats. Unfortunately, due to an implementation error, it fails to handle NaNs. This means that if you call *_mm_safeInt_ps<new_mm_ceil_ps2>()* with a NaN then you get a normal number back. Whenever possible NaNs should be 'sticky' in order to aid in tracking down the errors that produce them.

The problem is that the wrapper function uses *cmpgt* to create a mask that it can use to retain the value of large floats – this mask is all ones for large floats. However since all comparisons with NaNs are false this mask is zero for NaNs, so a garbage value is returned for them. If the comparison is switched to *cmple* and the two mask operations (*and* and *andnot*) are switched then NaN handling is obtained for free. Sometimes correctness doesn't cost anything. Here's a fixed version:

```
template< __m128 (FuncT)(const __m128&) >
inline __m128 _mm_fixed_safeInt_ps(const __m128& a){
    __m128 v8388608 = *(__m128*)&_mm_set1_epi32(0x4b0
    __m128 aAbs = _mm_and_ps(a, *(__m128*)&_mm_set1_e
    // In order to handle NaNs correctly we need to u
    // Using le ensures that the bitmask is clear for
    // NaNs, whereas gt ensures that the bitmask is s
    // but not for NaNs.
    __m128 aMask = _mm_cmple_ps(aAbs, v8388608);
    // select a if greater then 8388608.0f, otherwise
    // FuncT. Note that 'and' and 'andnot' were rever
    // meaning of the bitmask has been reversed.
    __m128 r = _mm_xor_ps(_mm_andnot_ps(aMask, a), _m
    return r;
}
```

With this fix and the latest version of *_mm_ceil_ps2* it becomes possible to handle all 4 billion floats correctly.

Conventional wisdom Nazis

Conventional wisdom says that you should never compare two floats for equality – you should always use an epsilon. Conventional wisdom is wrong.

I've written in great detail about how to compare floating-point values using an epsilon, but there are times when it is just not appropriate. Sometimes there really is an answer that is correct, and in those cases anything less than perfection is just sloppy.

So yes, I'm proudly comparing floats to see if they are equal.

How did the fixed versions do?

After the flaws in these functions were pointed out fixed versions of *_mm_ceil_ps2* and its sister functions were quickly produced and these new versions work better.

I didn't test every function, but here are the results from the final versions of functions that I did test:

- - XMVectorCeiling 3.06: zero failures
  - XMVectorFloor 3.06: zero failures
  - XMVectorRound 3.06: 33,554,432 errors on incorrectly handled boundary conditions
  - _mm_ceil_ps2 with _mm_safeInt_ps: 16777214 failures on NaNs
  - _mm_ceil_ps2 with _mm_fixed_safeInt_ps: zero failures
  - LiraNuna ceil: this function was not updated so it still has

864,026,625 failures.

Exhaustive testing works brilliantly for functions that take a single float as input. I used this to great effect when rewriting all of the CRT math functions for a game console some years ago. On the other hand, if you have a function that takes multiple floats or a double as input then the search space is too big. In that case a mixture of test cases for suspected problem areas and random testing should work. A trillion tests can complete in a reasonable amount of time, and it should catch most problems.

Test code

Here's a simple function that can be used to test a function across all floats. The sample code linked below contains a more robust version that tracks how many errors are found.

```
// Pass in a uint32_t range of float representations
// start and stop are inclusive. Pass in 0, 0xFFFFFFF
```

```
// floats. The floats are iterated through by increme
// their integer representation.
void ExhaustiveTest(uint32_t start, uint32_t stop, Tr
    Transform RefFunc, const char* desc)
{
    printf("Testing %s from %u to %u (inclusive).\n",
    // Use long long to let us loop over all positive
    long long i = start;
    while (i <= stop)
    {
        Float_t input;
        input.i = (int32_t)i;
        Float_t testValue = TestFunc(input.f);
        Float_t refValue = RefFunc(input.f);
        // If the results don't match then report an
        if (testValue.f != refValue.f &&
            // If both results are NaNs then we treat
            (testValue.f == testValue.f || refValue.f
        {
            printf("Input %.9g, expected %.9g, got %1
            input.f, refValue.f, testValue.f);
        }

        ++i;
    }
}
```

Subtle errors

My test code misses one subtle difference – it fails to detect one type of error. Did you spot it?

The correct result for ceil(-0.5f) is -0.0f. The sign bit should be preserved. The vector math functions all fail to do this. In most cases this doesn't matter, at least for game math, but I think it is at least important to acknowledge this (minor) imperfection. If the compare function was put into 'fussy' mode (just compare the representation

of the floats instead of the floats) then each of the *ceil* functions would have an additional billion or so failures, from all of the floats between -0.0 and -1.0.

References

The original post that announced *_mm_ceil_ps2* can be found here – with corrected code:

http://dss.stephanierct.com/DevBlog/?p=8

This post discusses the bugs in the 3.03 version of DirectXMath and how to get fixed versions:

http://blogs.msdn.com/b/chuckw/archive/2013/03/06/known-issues-directxmath-3-03.aspx

This post links to the LiraNuna glsl-sse2 math library:

https://github.com/LiraNuna/glsl-sse2

The original reddit discussion of these functions can be found here:

http://w3.reddit.com/r/programming/comments/1p2yys/sse3_optimized_vector_floor_ceil_round_and_mod/

Sample code for VC++ 2013 to run these tests. Just uncomment the test that you want to run from the body of *main*.

https://www.cygnus-software.com/ftp_pub/test4billion.zip

The reddit discussion of this post can be found here, and then here.

The hacker news discussion of this post can be found here, and then here. And again in 2020 it can be found here.

I've written before about running tests on all of the floats. The last time I was exhaustively testing round-tripping of printed floats, which took long enough that I showed how to easily parallelize it and then I verified that they round-tripped between VC++ and gcc.

This time the tests ran so quickly that it wasn't even worth spinning up extra threads.