Understanding fast float/integer conversions

If you are interested in micro-optimisation and have never studied IEEE-754 float representation, I suggest you have a look at the beast. It may give you ideas for interesting bitlevel operations. This article will cover the specific topic of conversions between integers and floats.

Note: unless you are coding for antique or very specific architectures such as the PDP-11, you may assume that the floating point storage endianness and the integer endianness match. The code presented here will therefore work flawlessly on modern CPU architectures such as x86, amd64, PowerPC or even ARM.

Introduction

Here are a few floating point values and their bitlevel representation. Notice how the different values affect the sign, exponent and mantissa fields:

	sign	exponent	mantissa
1.0f	0	01111111	0000000 00000000 00000000
-1.0f	**1**	01111111	0000000 00000000 00000000
0.5f	0	**01111110**	0000000 00000000 00000000
0.25f	0	**01111101**	0000000 00000000 00000000
1.0f + **0.5f**	0	01111111	**1000000** 00000000 00000000
1.0f + **0.25f**	0	01111111	**0100000** 00000000 00000000

The core idea behind this article is the manipulation of the last field, the mantissa.

Byte to float conversion

A classical byte (`0` - `255`) to float (`0.0f` - `1.0f`) conversion function is shown here:

```
float u8tofloat(uint8_t x)
{
    return (float)x * (1.0f / 255.0f);
}
```

This looks very simple: one conversion (`fild` on x86) and one multiplication (`fmul` on x86). However, the `fild` instruction has a latency such that the conversion may have a severe impact on performance.

But let's look at these interesting floating point values:

	sign	exponent	mantissa
32768.0f	0	10001110	0000000 00000000 **00000000**
32768.0f + **1.0f/256.0f**	0	10001110	0000000 00000000 **00000001**
32768.0f + **2.0f/256.0f**	0	10001110	0000000 00000000 **00000010**
32768.0f + **255.0f/256.0f**	0	10001110	0000000 00000000 **11111111**

Notice the last eight bits? They look almost exactly like the input byte expected by `u8tofloat`. Taking advantage of the binary representation allows us to write the following conversion function:

```
float u8tofloat_trick(uint8_t x)
{
    union { float f; uint32_t i; } u; u.f = 32768.0f; u.i |= x;
    return u.f - 32768.0f;
}
```

When used in a CPU-intensive loop, this method can be up to **twice as fast** as the previous implementation, for instance on the amd64 architecture. On the x86 architecture, the difference is far less noticeable.

You probably noticed that the output range is `0.0f` - `255.0f/256.0f` instead of `0.0f` - `1.0f`. This may be preferred in some cases when the value is supposed to wrap around. However, colour coordinates will require exact `0.0f` - `1.0f` bounds. This is easily fixed with an additional multiplication:

```
float u8tofloat_trick2(uint8_t x)
{
    union { float f; uint32_t i; } u; u.f = 32768.0f; u.i |= x;
    return (u.f - 32768.0f) * (256.0f / 255.0f);
}
```

This can still be up to twice as fast than the original integer to float cast.

Short to float conversion

The usual way to convert a 16-bit integer to a float will be:

```
float u16tofloat(uint16_t x)
{
    return (float)x * (1.0f / 65535.0f);
}
```

Again, careful observation of the following floats will be useful:

	sign	exponent	mantissa
16777216.0f	0	10010111	0000000 **00000000 00000000**
16777216.0f + **1.0f/65536.0f**	0	10010111	0000000 **00000000 00000001**
16777216.0f + **2.0f/65536.0f**	0	10010111	0000000 **00000000 00000010**
16777216.0f + **65535.0f/65536.0f**	0	10010111	0000000 **11111111 11111111**

And the resulting conversion method:

```
float u16tofloat_trick(uint16_t x)
{
    union { float f; uint32_t i; } u; u.f = 16777216.0f; u.i |= x;
    return u.f - 16777216.0f; // optionally: (u.f - 16777216.0f) * (65536.0f / 65535.0f)
}
```

However, due to the size of the input data, the performance gain here can be much less visible. Be sure to properly benchmark.

Int to float conversion

The above techniques cannot be directly applied to 32-bit integers because floats only have a 23-bit mantissa. Several methods are possible:

- Use the `double` type instead of `float`. They have a 52-bit mantissa.
- Reduce the input `int` precision to 23 bits.

Float to int conversion

Finally, the exact same technique can be used for the inverse conversion. This is the naive implementation:

```
static inline uint8_t u8fromfloat(float x)
{
    return (int)(x * 255.0f);
}
```

Clamping is left as an exercise to the reader. Also note that a value such as `255.99999f` will ensure better distribution and avoid singling out the `1.0f` value.

And our now familiar bitlevel trick:

```
static inline uint8_t u8fromfloat_trick(float x)
{
    union { float f; uint32_t i; } u;
    u.f = 32768.0f + x * (255.0f / 256.0f);
    return (uint8_t)u.i;
}
```

Unfortunately, this is usually a performance hit on amd64. However, on x86, it is up to **three time as fast** as the original. Choose wisely!

The LUT strategy

Some will point out that using a lookup table is much faster.

```
float lut[256];
void fill_lut()
{
    for (int n = 0; n < 256; n++) lut[n] = (float)n / 255.0f;
}
float u8tofloat_lut(uint8_t x)
{
    return lut[x];
}
```

This is indeed faster in many cases and should not be overlooked. However, the following should be taken into account:

- LUTs are fast, but if unlucky, the cache may get in your way and cause performance issues
- the LUT approach is actually *almost always slower* with 16-bit input, because the size of the table starts messing with the cache
- do not underestimate the time needed to fill the LUT, especially if different conversions need to be performed, requiring several LUTs
- LUTs do not mix well with SIMD instructions
- obviously, this method doesn't work with float to int conversions

Last warnings

Many programmers will be tempted to write shorter code such as:

```
float u8tofloat_INVALID(uint8_t x)
{
    // NO! DON'T DO THIS! EVER!
    float f = 32768.0f; *(uint32_t *)&f |= x;
    return f - 32768.0f;
}
```

Do not do this, ever! I *guarantee* that this will break in very nasty and unexpected places. Strict C and C++ aliasing rules make it illegal to have a pointer to a float also be a pointer to an integer. The only legal way to do this is to use a `union` (actually, this is still not legal by the C++ standard but most real-life compilers allow this *type punning* through documented extensions).

Finally, one last, obvious tip: always measure the effects of an optimisation before deciding to use it!