

Fast Half Float Conversions

Jeroen van der Zijp
November 2008

Introduction.

High dynamic range imaging and signal processing require more compact floating point representations than single precision (32-bit) IEEE 754 standard allows. To meet these objectives, a 16-bit “half” float data type was introduced. Hardware support for these is now common place in Graphics Processing Units (GPU's), but unfortunately not yet in CPU's.

Because of this, calculations using 16-bit half-floats must be done using regular 32-bit IEEE floats, necessitating frequent conversions from half-floats to floats and vice-versa.

Half Float Representations.

The half-float data type is inspired by the IEEE 754 standard, except sacrifices range and accuracy in favor of representation size. A half-float comprises a sign bit, a 5-bit exponent with a bias of 15, and a 10-bit mantissa, see Figure 1 below.

| | | | | | | | | | | | | | | | |
|----|----------|----|----|----|----|----------|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| s | exponent | | | | | mantissa | | | | | | | | | |

Figure 1. Half Float Representation

Interpretation of the half float representation is as follows:

- If the exponent field is in the range [1..30], then the value represented is:

$$\text{value} = (-1)^s \cdot 2^{(\text{exponent}-15)} \cdot 1.\text{mmmmmmmmmm}$$

This is the case for normalized half-float numbers.

- If the exponent field is 0 (zero), and the mantissa is not zero:

$$\text{value} = (-1)^s \cdot 2^{-14} \cdot 0.\text{mmmmmmmmmm}$$

In this particular case, the number is called subnormal (denormal), and has less accuracy in its mantissa.

- If the exponent field is zero, and the mantissa is also zero:

$$\text{value} = \pm 0.0$$

- If the exponent value is 31, and the mantissa is 0 (zero):

$$\text{value} = \pm\infty \quad (\text{Infinity})$$

- Finally, if the exponent is 31 and the mantissa is not zero:

$$\text{value} = \pm\text{NaN} \quad (\text{Non a Number})$$

Conversion Requirements.

When performing calculations using half-floats numbers, the numbers must be converted to floats first, and then back to half-floats. Consequently, these conversions must be very fast. Also, it would be nice if a conversion from half-float to float and then back to half-float would yield the original number. Finally, special cases like subnormal numbers, infinity, and NaNs should be handled properly.

The IEEE 754 float representation, shown in Figure 2, is the format to/from which the half-floats are to be converted. The IEEE 754 float comprises a sign bit, an 8-bit exponent with a bias of 128, and a 23-bit mantissa.

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|----|----|----|----|----|----|----|----------|----|----|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| s | exponent | | | | | | | | mantissa | | | | | | | | | | | | | | | | | | | | | | |

Figure 2. IEEE 754 Float Representation.

Conversion of Half Float to Float.

Conversion of half float to float is, in principle, simple: copy the sign bit, subtract the half-float bias (15) from the exponent and add the single-precision float bias (127), and append 13 zero-bits to the mantissa. In C code:

$$f = ((h \& 0x8000) \ll 16) \mid (((h \& 0x7c00) + 0x1c000) \ll 13) \mid ((h \& 0x03ff) \ll 13)$$

The above expression however only works for “normal” half-float numbers; it does not work for special cases such as zero or subnormal (denormal) numbers.

A correct conversion must treat all the special cases:

1. Zero (and negative zero).
2. Subnormal half-float values should map to their corresponding float values; since floats have a larger range than half-floats, subnormal half-float values will map to proper float values.
3. The half float value *Infinity* should map to a float value of *Infinity*.
4. Half float NaN (not a number) values should map to float NaN values.

A simple and correct algorithm for the conversion would be to pre-calculate all possible conversion values. There are only 65536 of them. Unfortunately, that is still a very large table (262,144 bytes to be exact).

Fortunately, it is possible to reduce the size of the table by exploiting some patterns in the lookup process:

- Except for the case where the exponent of the half-float is 0, the mantissa is always simply

- padded with zeros.
- When the exponent of the half-float is 31 (the number represents either Infinity or NaN), the resulting float number must have an exponent value of 255.
- When the exponent of the half-float is zero and the mantissa is non-zero, the number is a subnormal half-float. These values can be represented as normalized float by adjusting exponent and shifting the mantissa appropriately.

At the expense of a few extra operations, the original direct lookup method can be reimplemented with substantially smaller tables:

```
f=mantissatable[offsettable[h>>10]+(h&0x3ff)]+exponenttable[h>>10]
```

In the above, the **mantissatable** is only 2028 entries, and the **offsettable** and **exponenttable** are only 64 entries each. The total amount of space required is thus only 8576 bytes [the **offsettable** requires only 2 bytes per entry], a mere 3.2% of the original table size. This much smaller table is more likely to fit into modern processor caches and thus this algorithm is expected to run faster than the original simple lookup process due to faster memory accesses, despite performing a few more operations.

The **mantissatable** is populated as follows:

```
mantissatable[0] = 0
mantissatable[i] = convertmantissa(i)           for i = 1..1023
mantissatable[i] = 0x38000000 + ((i-1024)<<13)  for i = 1024...2047
```

The **exponenttable** is set up as below:

```
exponenttable[0] = 0,
exponenttable[32]= 0x80000000

exponenttable[i] = i<<23           for i = 1..30
exponenttable[i] = 0x80000000 + (i-32)<<23  for i = 33..62

exponenttable[31]= 0x47800000
exponenttable[63]= 0xC7800000
```

And finally, the **offsettable** is filled as below:

```
offsettable[0] = 0
offsettable[32]= 0

offsettable[i] = 1024           everywhere else.
```

The function `convertmantissa()` transforms the subnormal representation to a normalized one, as follows:

```
unsigned int convertmantissa(unsigned int i){
    unsigned int m=i<<13;           // Zero pad mantissa bits
    unsigned int e=0;               // Zero exponent
    while(!(m&0x00800000)){         // While not normalized
```

```

        e-=0x00800000;           // Decrement exponent (1<<23)
        m<<=1;                   // Shift mantissa
    }
    m&=~0x00800000;             // Clear leading 1 bit
    e+=0x38800000;             // Adjust bias ((127-14)<<23)
    return m | e;              // Return combined number
}

```

Description of the algorithm. Essentially, the conversion of a half-float to a float is written as the conversion of the mantissa plus the conversion of the exponent and the sign. There are essentially three cases. In the subnormal case, the half-float exponent is zero. The half-float mantissa is normalized and the exponent is adjust accordingly. This calculation was performed by `convertmantissa()` when the table was generated. We only need to add the sign.

In the normal case, the half-float exponent is non-zero, and we simply shift the mantissa left over 13 bits and add the exponent bias adjustment $(127-15)\ll 23$, or `0x38000000`. To this, the value of the exponent is simply added. Finally, in the case of Infinity or NaN, the mantissa is treated as before, except now an extra adjustment needs to be added $(255-(127-15))\ll 23$, or `0x47800000` to obtain the proper exponent value for Infinite or NaN numbers in the float representation.

Timings have been performed on a 2GHz Core² Duo Intel CPU, yielding typically about 5.6ns for this conversion.

Conversion of Float to Half-Float.

Intuitively, conversion from float to half-float is a slightly more complex process, due to the need to handle overflows and underflows. As before, there is a compact and simple version which is pretty straight-forward:

```
h = ((f>>16)&0x8000) | (((f&0x7f800000)-0x38000000)>>13)&0x7c00 | ((f>>13)&0x03ff)
```

In a nutshell, first the sign bit is shifted and masked, then the exponent is masked off, and the bias-correction subtracted; the result is shifted, and finally the mantissa is shifted and masked off. All the pieces are then assembled together.

The above expression suffers from a few fatal flaws, however. It doesn't handle zero, Infinity, NaN, or small float numbers which are only representable as subnormal half-floats.

Proper conversion must handle the following cases:

1. Really small numbers and zero should map to a half-float value of zero.
2. Small numbers should convert to subnormal half-float values.
3. Regular magnitude numbers should just lose some precision.
4. Large numbers should map to half-float Infinity.
5. Infinity and NaNs should remain Infinity and NaNs in their half-float representation.

Ideally, it would be nice if the conversions were round-trip safe, i.e. half-float \rightarrow float \rightarrow half-float would yield the same identical number one started out with.

Fortunately, extremely fast conversion can again be implemented using a purely table-driven approach.

The algorithm is as follows:

```
h=basetable[(f>>23)&0x1fff]+((f&0x007fffff)>>shifftable[(f>>23)&0x1fff])
```

The algorithm uses only two small tables, each with 512 entries only. The **basetable** is only 1024 bytes, while the **shifftable** is only 512 bytes, yielding a total of 1536 bytes.

The tables are populated using a small program, although in an actual deployment these tables are more likely fixed at compile time:

```
void generatetables(){
    unsigned int i;
    int e;
    for(i=0; i<256; ++i){
        e=i-127;
        if(e<-24){ // Very small numbers map to zero
            basetable[i|0x000]=0x0000;
            basetable[i|0x100]=0x8000;
            shifftable[i|0x000]=24;
            shifftable[i|0x100]=24;
        }
        else if(e<-14){ // Small numbers map to denorms
            basetable[i|0x000]=(0x0400>>(18-e));
            basetable[i|0x100]=(0x0400>>(18-e)) | 0x8000;
            shifftable[i|0x000]=-e-1;
            shifftable[i|0x100]=-e-1;
        }
        else if(e<=15){ // Normal numbers just lose precision
            basetable[i|0x000]=((e+15)<<10);
            basetable[i|0x100]=((e+15)<<10) | 0x8000;
            shifftable[i|0x000]=13;
            shifftable[i|0x100]=13;
        }
        else if(e<128){ // Large numbers map to Infinity
            basetable[i|0x000]=0x7C00;
            basetable[i|0x100]=0xFC00;
            shifftable[i|0x000]=24;
            shifftable[i|0x100]=24;
        }
        else{ // Infinity and NaN's stay Infinity and NaN's
            basetable[i|0x000]=0x7C00;
            basetable[i|0x100]=0xFC00;
            shifftable[i|0x000]=13;
            shifftable[i|0x100]=13;
        }
    }
}
```

Description of the algorithm. There are five cases, each case solely identified by the exponent value of the incoming number.

- When the magnitude of the number is really small (2^{-24} or smaller), there is no possible half-float representation for the number, so it must be mapped to zero. This is accomplished by setting the **basetable** entries to zero (or zero with sign). The **shifftable** entries are set to 24, meaning all the bits of the original mantissa are shifted out completely, leaving just zero.

- When the number is small (smaller than 2^{-14}), the value can only be represented using a subnormal half-float. This is the most complex case: first, the leading 1 bit, implicitly represented in the normalized representation, must be explicitly added, and then the resulting mantissa must be shifted rightward, over a number of bit-positions as determined by the exponent. In the above implementation however, we prefer to shift the original mantissa bits, and add the pre-shifted 1-bit to it. Thus, the **basetable** entry is filled with the expression:

`basetable[i] = 0x0400>>(18-e), where e=i-127 and -24<=e<-14.`

As before, for negative numbers a sign-bit (0x8000) is also added to the **basetable** entry.

- Normal numbers (smaller than 2^{15}), can be represented using half-floats, albeit with slightly less precision. The entries in the basetable are simply set to the bias-adjust exponent value, shifted into the right position. A sign bit is added for the negative case.
- Large float values (numbers less than 2^{128}) must be mapped to half-float Infinity. They are too large to be represented as half-floats. In this case the basetable must be set to 0x7C00 (with sign if negative) and the mantissa must be zeroed out, which is accomplished by shifting out all mantissa bits.
- Finally, remaining float numbers, Infinity and NaNs should stay Infinity and NaNs after half-float conversion. The basetable entry is exactly the same as for the previous case, except the mantissa-bits are to be preserved as much as possible, thus ensuring that Infinity stays infinite and NaN's stay NaN's.

Timings have been performed on this algorithm, again on a 2GHz Core² Duo Intel CPU. Typical values are around 3.25ns for each conversion.

Conclusion.

Extremely fast half-float to float and float to half-float conversion algorithms have been introduced. They are branch-free, table-driven, using fairly compact tables which are easily fitted into primary caches on modern CPUs.

Contrary to expectation, conversion from half-float to float takes a bit longer than the other way round, at least on the processors tested. Since both forward and backward conversion are very similar in terms of number of operations, this difference is largely attributed to the fact that the lookup tables in the half-float to float case are much larger, and perhaps may reside partially in L2 cache instead of fitting completely in L1 cache.

Either way, these new conversions are very fast, and represent a substantial improvement over previous known algorithms, both in speed as well as storage requirements.