Appendix A

Fixed-point Mathematics

In this appendix, we will introduce the notation and operations that we use for fixed-point mathematics. For some platforms, e.g., low-cost mobile phones, fixed-point mathematics is used on the CPU due to lack of accelerated floating point instructions. It can be argued that floating point acceleration on the CPU will be added in the future, and so fixed-point math is nothing that is relevant to learn. However, in the hardware of a GPU, it often makes sense to use fixed-point math. To reduce the cost of an implementation, it is very important to reduce the needed accuracy (number of bits) in the computations without sacrificing quality.

It should also be pointed out that fixed-point math can be more accurate than floating point, especially if you know the maximum range of your numbers before starting.

A.1 Notation

In contrast to floating point, a fixed-point number has a decimal point with fixed position. Each number has a number of bits for the integer part, and another number of bits for the fractional part. With i integer bits, and f fractional bits, the notation for that representation is shown in Equation A.1.

$$[i.f] \tag{A.1}$$

Note that the most commonly used technique is to use two-complement if negative numbers are to be represented as well. Thus, there is no explicit sign bit as in floating point. The *resolution* of a fixed-point number is dictated by the number of fractional bits, f, and the resolution is 2^{-f}. This is the smallest unit that can be handled using this representation.

For f bits for the fractional part, the resolution is shown in Table A.1.

f	Resolution	Resolution
1	1/2	0.5
2	1/4	0.25
3	1/8	0.125
4	1/16	0.0625
5	1/32	0.03125
6	1/64	0.015625
7	1/128	0.0078125
8	1/256	0.00390625
12	1/4,096	0.000244140625
16	1/65,536	0.0000152587890625
24	1/16,777,216	0.00000059604644775390625
32	1/4,294,967,296	0.0000000023283064365386962890625

Table A.1: Resolution of fixed-point numbers with f fractional bits. The resolution is shown both as a rational number ($1/2^f$), and a decimal number.

A.1.1 Conversion

Converting between fixed-point and floating-point numbers is simple as shown in the examples below.

Example A.1.1.1 Conversion from floating-point to fixed-point
For example, assume you have a floating-point number, 2.345. Converting it to the $[5, 4]$-format is done as follows: $\texttt{round}(2.345 \times 2^4) = 38$. □

Example A.1.1.2 Conversion from fixed-point to floating-point
Now assume that we have a fixed-point representation in $[5, 4]$-format, and that the integer value of that representation is 38 (see example above). Converting it back to floating-point is done as follows: $\texttt{float}(38 \times 2^{-4}) = 2.375$. If truncation (using for example `int()` in C/C++) was used instead of `round()` in Example A.1.1.1, the fixed-point representation would become 37 instead, and when converting back to float the result would be 2.3125, which is a slightly worse approximation compared to 2.375. □

As can be seen in the two examples above, $2.345 \neq 2.375$, and this is one of the obstacles in using fixed-point representation: the accuracy can be quite bad, and therefore, extreme care has to be taken in order to provide sufficient accuracy. Usually, the indata and the needed accuracy in the result are analyzed, and fixed-point format is chosen thereafter.

In general, converting a floating-point number, a, to fixed point format, $[i.f]$, is done as shown below:

$$\texttt{round}(a \times 2^f), \tag{A.2}$$

where the result is an integer with $i + f$ bits. The corresponding formula for converting from fixed-point, b, to floating-point is:

$$\texttt{float}(b \times 2^{-f}) \tag{A.3}$$

Note that conversion therefore often is implemented as shifting the numbers, since this is less expensive than multiplication and division.

It should be noted that we are not restricted to using scaling factors of 2^f and 2^{-f}. In general, any number can be used as scaling factor, but the powers of two are convenient because they can be implemented as simple left and right shifts. Furthermore, using scaling factors that are not powers of two makes for a representation that is not on $[i.f]$-format.

Rounding of a floating-point number, a, can be implemented as follows:

$$\texttt{round}(a \times 2^f) = \texttt{int}(\lfloor a \times 2^f + 0.5 \rfloor), \tag{A.4}$$

where `int()` is a function that simply truncates the fractional parts, i.e., throws them away. The floor function, $\lfloor x \rfloor$ simply truncates towards $-\infty$, e.g., $\lfloor 1.75 \rfloor = 2$ and $\lfloor -1.75 \rfloor = -2$.

A.2 Operations

In the following, we show what happens to the resulting fixed-point representation that is need to exactly being able to represent the result from an operation, e.g., multiplication. Such knowledge can help prevent overflow/underflow from occurring.

A.2.1 Addition/Subtraction

Addition and subtraction are performed by treating the fixed-point numbers as integers and adding them using standard addition/subtraction. In terms of the bit count of the result, that can be expressed as shown below:

$$[i.f] \pm [i.f] = [i + 1.f]. \tag{A.5}$$

As can be seen, all that happens is that the number of integer bits are increased by one. Intuitively, this makes sense, since adding the biggest number that can be represented in $[i.f]$ to itself, is the same as multiplying the number by two, which in turn is the same as shifting the number one step to the left. This always makes the number occupy one more bit, and correspondingly, the number of integer bits has been increased by one $(i + 1)$.

Example A.2.1.1
Adding two numbers represented in [8.8]-format gives a sum that is represented in [9.8], and hence, the result cannot be stored in 16 bits. □

If the two operands in the addition/subtraction has different number of bits, the bit count of the result is expressed as:

$$[i_1.f_1] \pm [i_2.f_2] = [\max(i_1, i_2) + 1.\max(f_1, f_2)], \tag{A.6}$$

that is, the number of integer bits of the results is one plus the maximum of the integer bits of the operands, and the fractional number of bits is equal to

the maximum number of bits of the operands' fractional bits. However, in that case care must be taken so that standard integer addition can be used. This is done by aligning the number representations, as shown in the example below.

Example A.2.1.2
Assume we have two fixed-point numbers $a = [8.8]$ and $b = [4.4]$, and that we want to compute $c = a + b$. Furthermore, we assume that a and b are standard integers, and we want to add the numbers using standard addition. To make that work, the numbers must be aligned, and that is done by making sure they have the same number of fractional bits. Hence, we must see to it that b also has 8 fractional bits. This is done by simply shifting b four steps to the left. Therefore, the computation is done as $c = a + (b << 4)$, where c is on [9.8]-format. □

A.2.2 Multiplication

Multiplying two numbers is a bit more complicated. In terms of the number of bits in the result, Equation A.7 shows what happens.

$$[i.f] \times [i.f] = [2i.2f] \tag{A.7}$$

Multiplication of two fixed points numbers are performed as standard integer multiplication. However, as can be seen above, the number of fractional (and integer) bits have doubled. Thus, the decimal point is no longer at the same position.

The factor two comes from that multiplication can be thought of as a series of additions. For simplicity, consider an integer with i bits. The biggest number, h, is again when all bits are set to one. Multiplying this number by itself, $h \times h$, gives us the biggest possible product. This product can be expressed as: $h \times h = h + 2h + 4h + \cdots + 2^{i-1}h$. That is, first we have i bits in h, and we add $2h$, and that sum is represented using $i + 2$ bits according to Equation A.6. Then we add $4h$, which is represented with $i + 2$ bits as well. Thus, $h + 2h + 4h$ can be represented using $i + 3$ bits, and so on. After all terms have been added, this has grown to exactly i extra bits since $i - 1$ additions have to be done, and because another one is added due to addition. Hence the factor two in Equation A.7. Similar reasoning can be applied to the fractional bits as well.

In the general case, the resulting product is computed as shown below:

$$[i_1.f_1] \times [i_2.f_2] = [i_1 + i_2.f_1 + f_2]. \tag{A.8}$$

Again, one has to be careful when considering where the decimal point is located.

Example A.2.2.1 Multiplication of fixed-point numbers I
Assume that two fixed-point numbers [2.0] are to be multiplied. These numbers do not have any fractional bits, and only two integer bits. The biggest numbers that can be represented are 11_b, and the biggest product that can be produced is then $11_b \times 11_b = 1001_b$. As can be seen, the result has four bits as predicted. □

Example A.2.2.2 Multiplication of fixed-point numbers II
Assume we have two floating-point numbers $r_1 = 0.79$ and $r_2 = 3.15$ that shall be multiplied using fixed-point representation [4.5]. Converting to fixed-point gives: $f_1 = \texttt{round}(0.79 \times 2^5) = 25$ and $f_2 = \texttt{round}(3.15 \times 2^5) = 101$. Multiplying gives: $f_3 = f_1 \times f_2 = 25 \times 101 = 2525$. Recall, that f_3 must be on $[4 + 4.5 + 5] = [8.10]$-format. Thus, when converting back to floating-point, we get: $r_3 = \texttt{float}(2525 \times 2^{-10}) = 2.465820\ldots$. The real result should be $0.79 \times 3.15 = 2.4885$. Note, that with more fractional bits in the initial conversion from floating-point to fixed-point, the end result will be more accurate. □

A.2.3 Reciprocal

Recall that the resolution of a fixed-point number using $[i.f]$-representation is 2^{-f}, that is, the smallest number that we can represent is the same as the resolution. When computing the reciprocal, $1/x$, then the biggest result that can be obtained must occur when the denominator, x, is smallest. Thus, the biggest number obtained through computing a reciprocal of a fixed-point number is $1/2^{-f} = 2^f$, and clearly there need to be f integer bits to represent this number. Hence, the result of computing $1/x$, where x is on $[i.f]$-format, must be on $[f.z]$-format. This means that in general, for reciprocal computation, it holds that:

$$\frac{1}{[i.f]} = [f.z]. \tag{A.9}$$

As can be seen in the in the following example, z, can, unfortunately, be infinitely large.

Example A.2.3.1 Reciprocal of fixed-point numbers
The fixed-point representation of 0.75 using three fractional bits is $\texttt{round}(0.75 \times 2^3) = 6$. The reciprocal is then $1/6 = 0.16666666666667$, but with infinitely many decimals. Clearly, this is not feasible to represent exactly using fixed-point. □

A reasonable representation of the reciprocal can be obtained by considering which is the biggest number that can be represented using $[i.f]$-format. Assuming we use signed fixed-point, the largest integer we can represent, using the i integer bits, is $2^i - 1$, and the fractional bits must represent a number, b, which is less than one, namely, $b \leq 1 - 2^{-f}$. Thus, the sum of these is $2^i - 1 + b \leq 2^i - 1 + 1 - 2^{-f} < 2^i$. If we assume that the number is exactly 2^i, the reciprocal will be 2^{-i}, that is, i fractional bits are needed. Hence, we have the following approximation:

$$\frac{1}{[i.f]} \approx [f.i]. \tag{A.10}$$

A more pragmatic way of thinking about this, is to decide which accuracy you need in the resulting number. The accuracy of the reciprocal is then:

$$\frac{\texttt{round}(1 \times 2^{f_1})}{[i_2.f_2]} = [f_2.f_1 - f_2], \tag{A.11}$$

where the nominator is a fixed-point representation of 1 with f_1 fractional bits, and $f_1 \geq f_2$.

Example A.2.3.2
Assume that you want to compute the reciprocal of x and that you want the resulting number to have $f = 11$ fractional bits, because that is needed for subsequent computations. Now, if x has $f_2 = 5$ fractional bits, then the nominator in Equation A.11 must have $f_1 = 11 + 5$ bits, since that gives a result with $f_1 - f_2 = 16 - 5 = 11$ bits.

A.2.4 Division

TODO: This part of these notes is not finished, so the students can avoid this subsection. It is not needed in the course anyway, right now. □
Division:

$$\frac{[i_1.f_1]}{[i_2.f_2]} \approx [i_1 + f_2.f_1 - f_2], \tag{A.12}$$

where again $f_1 \geq f_2$.

A.2.5 Inexpensive division: special cases

For certain cases, computing the division by computing the reciprocal first, and then multiplying by that may be simpler. The special case of dividing by $2^n - 1$ is described by Blinn [12]. Looking at the fixed-point representation of such numbers, we find the following:

$$\begin{aligned} 1/3 &= 0.010101010101010\ldots \\ 1/7 &= 0.001001001001001\ldots \\ 1/15 &= 0.000100010001000\ldots \\ 1/31 &= 0.000010000100001\ldots \end{aligned}$$

As can be seen, the number of zeroes between the ones are becoming more and more, the higher the denominator. An approximation to $a/3$ can thus be implemented as $(a \times \texttt{5555}_x) >> 16$. For $1/255$, there are seven zeroes followed by a one, followed by seven zeroes and a one, and so on. Using 16 fractional bits, we can have the following approximation:

$$\frac{1}{255} \approx 1 << 8 + 1, \tag{A.13}$$

where the righthand side is on [0.16]-format. Thus, approximative division by 255 can be done by a shift and an add (and possibly some further shifting to get the result in the desired format). This is very inexpensive, compared to an integer division.

The original motivation [12] for such a division, was to compute the product of two eight-bit numbers, where 0 represented 0.0 and 255 represented 1.0. Assume we want to compute $c = a \times b$, where the result, c, is in the same format

as the terms, a and b, in the product. This is done by computing $a \times b/255$. Assuming $d = a \times b$ (where d uses 16 bits), the trick above can be used to compute c as:

$$c = ((d << 8) + d) >> 16. \tag{A.14}$$

Now, since we wanted the result, c, to be stored in eight bits, there is yet another optimization to be done. Note that since d uses 16 bits, $d << 8$ must use 24 bits. Performing addition with more bits costs more if you are to build hardware for it, and therefore, it would be nice to reduce the size of the computations as much as possible.

Now, assume that $d = \texttt{HHLL}_x$, that is a 16-bit word (as above), with a higher-order byte, $\texttt{HH}_x$, and a lower-order byte, $\texttt{LL}_x$. Equation A.14 can be illustrated as follows:

```
 HHLL00        (d << 8)
+  HHLL        d
-------
 ssssLL
```

If we only need the eight significant bits of this sum, then the least significant bits can be safely ignored, since they cannot carry any information over to the rest of the sum (due to that the first operand as $\texttt{00}_x$ as the least significant bits). Equation A.14 can therefore be rewritten as:

$$c = (d + (d >> 8)) >> 8 \tag{A.15}$$

The nice property about this, is that the hardware for Equation A.15 is simpler than that for Equation A.14 since 24 bits plus 16 bits has been replaced by 16 bits plus 8 bits.

If you want include correct rounding as well, then Blinn [12] argues that it should be done like this:

$$c = (d + (d >> 8) + 1) >> 8. \tag{A.16}$$

A.2.6 Expansion of integers and fixed-point numbers

We start this section by a simple example that serves as motivation for how the expansion is done in general later on.

Example A.2.6.1 Integer expansion: from four to eight bits
Assume we have a four-bit integer, c_4, that we want to expand to eight bits in order to obtain c_8. The biggest number we can represent in four bits is $2^4 - 1 = 15$, and in eights bits, it is $2^8 - 1 = 255$. Thus, c_4 should be multiplied by $255/15 = 17$ in order to expand c_4 into c_8. This can be done like this: $c_8 = 17 \times c_4 = 16 \times c_4 + c_4 = (c_4 \ll 4)$ `or` c_4, and this result is bit exact. □

In general, we do not get an integer when we compute $2^{i_1}/2^{i_2}$, $i_1 > i_2$, as we did in the example shown above, and therefore, we cannot get an exact expansion

in the majority of cases. However, we can get a very good approximation using a trick that resembles what we did in Example A.2.6.1. Assume we have an integer, c, using i_2 bits that should be expanded into d using i_1 bits. This can be done as shown in Equation A.17.

$$\begin{aligned} d &= c \ll (i_1 - i_2) + c \gg (i_2 - (i_1 - i_2)) = \\ &= c \ll (i_1 - i_2) \text{ or } c \gg (2i_2 - i_1) \end{aligned} \tag{A.17}$$

Put another way, the original number is first shifted to the left so it occupies all the i_1 bits, and then we need to fill out the least significant bits of the number, and those are filled out using the most significant bits from c.

Example A.2.6.2 Integer expansion
Assume we have an integer in binary format, 10110_b, using five bits. To expand this to eight bits, we compute $(10110_b \ll 3) + (10110_b \gg 2) = 10110000_b$ or $101_b = 10110101_b$. Thus, to expand $10110_b = 22$ into eight bits, we get $10110101_b = 181$. The biggest number we can represent with five bits is $2^5 - 1 = 31$. To convert to eight bits, we would ideally like to multiply by $255/31 = 8.2258...$ Our expansion did a good job though, since the expansion is equivalent to a multiplication by $181/22 = 8.2272...$, which is far better than multiplying by eight. □

a