Optimizing Exp/Log

For SwiftShader

Nicolas Capens <capn@google.com>

Introduction

SwiftShader is a conformant implementation of the Vulkan graphics API which runs entirely on the CPU. We've identified transcendental functions, and in particular the exponential and logarithm, to require optimization.

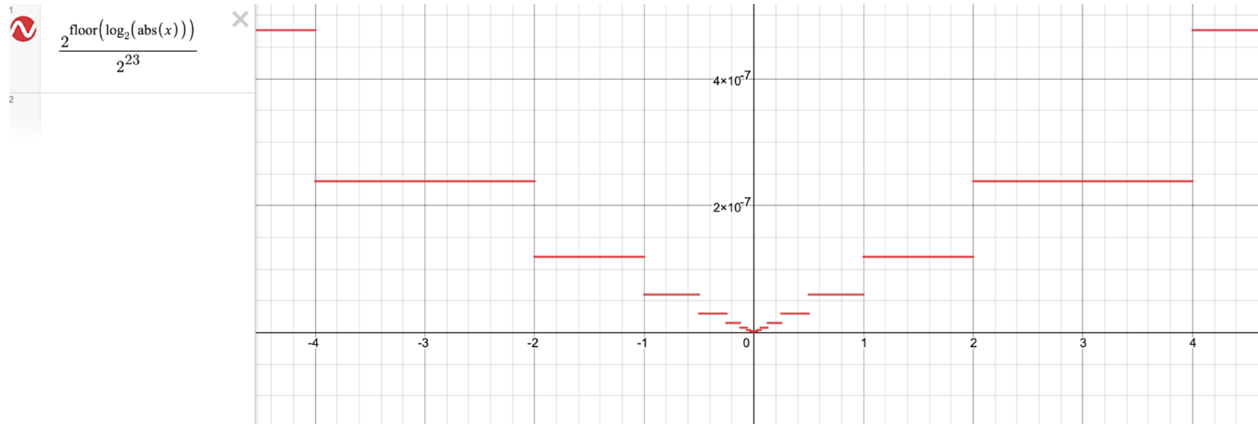
Objective

Vulkan specifies that the single-precision (32-bit) result of the `exp(x)` and `exp2(x)` shader operations must have an accuracy of $3 + 2 \times |x|$ ULP (note that Vulkan uses precision and accuracy interchangeably). `log(x)` and `log2(x)` have similar tolerances.

The new implementations will also have to be vectorizable to take advantage of wide SIMD.

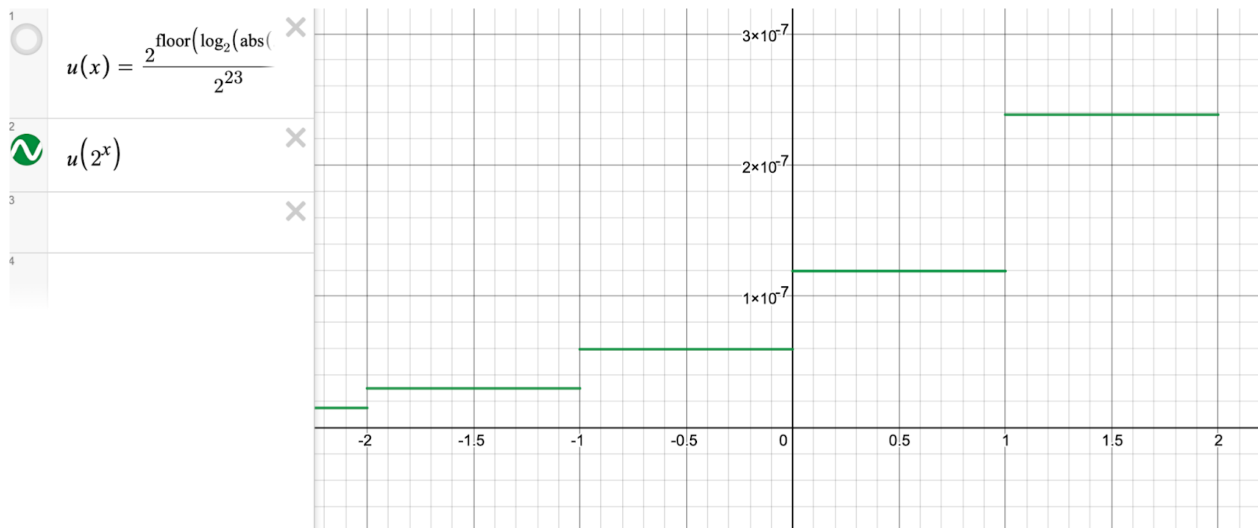
ULP: Discontinuous Relative Error Measure

Before we dive into any implementation details, we need to fully understand the requirements. Vulkan defines the error tolerance for `exp2` and `log2` in terms of ULP, or Units in the Last Place. 1 ULP is essentially the smallest representable difference in floating-point values. Note that this is relative to the value we're comparing against. Hence it is not a constant, but a function. It is convenient for a specification to drop the function parameter and just imply it. That said, I'll use ULP when talking about the error at a given point or the maximum error over an interval, and use $ulp(x)$ to denote the function. The image below, produced using the excellent Desmos graphing calculator, illustrates $ulp(x)$ for a single-precision (32-bit) value:

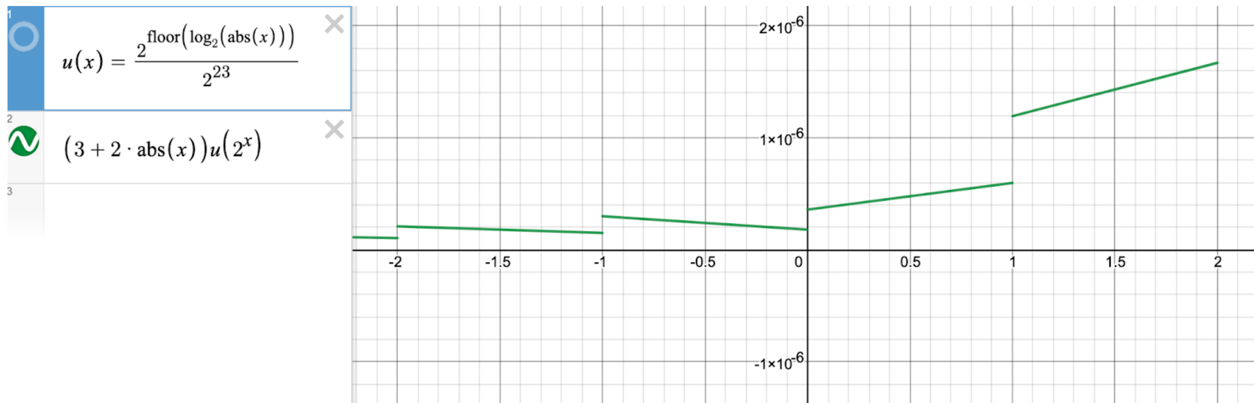


It is discontinuous due to the last manissa bit having a 2× difference in numerical value between every consecutive exponent value. Note that it is a fractal; zooming closer to 0 gives us ever smaller line segments 2× shorter but also 2× lower in y-axis value. That said, we can observe that $ulp(x)$ is roughly linear in x. Therefore by first approximation it can be considered a measure of the relative error.

Since the ULP is relative to the *result* of a function, we have to use that function as the input to the $ulp(x)$ function. For example for $exp_2(x)$ we get $ulp(exp_2(x))$:



Note how for x in $[0, 1)$, 1 ULP is a constant 2^{-23}. In other words, an absolute error! However, Vulkan tolerates an error of $3 + 2 \times |x|$, and when we take that into account we get our final error bounds:



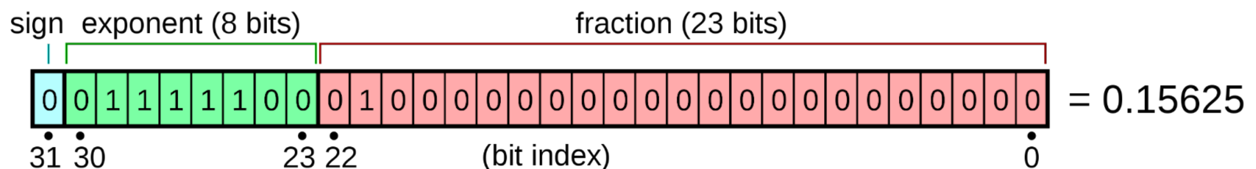
Globally this still curves like 2^x, but locally these are sloped linear segments.

Finally, note that ULP is also a function of the precision of the floating-point format we're using (or comparing it against). A single-precision `float` has 23 mantissa bits, while a 64-bit `double` has 53 mantissa bits. Then there's also 16-bit half-precision, which typically has 10 mantissa bits. Note that one can store a half-precision value in a 32-bit float, or evaluate the ULP error of a 32-bit value more precisely by comparing it against a value computed in 64-bit. Where useful, I'll discern between these precisions as ULP-32 and ULP-16.

Exp2 Range Reduction

The mathematical identity $exp_b(x_1 + x_2) = exp_b(x_1) \cdot exp_b(x_2)$ lends itself for computing the exp_2 of the integer and the fractional part of the argument separately. That is, $exp_2(x_i + x_f) = exp_2(x_i) \cdot exp_b(x_f)$, where $x_i = floor(x)$ and $x_f = x - x_i$. The exp_2 of the integer x_i is trivial to compute: just put it in the exponent field of the binary representation of a floating-point value (adjusted for the bias), with mantissa 0.

The IEEE 754 binary representation of a single-precision floating-point number consists of a sign bit, an 8-bit exponent field, and a 23-bit mantissa field:

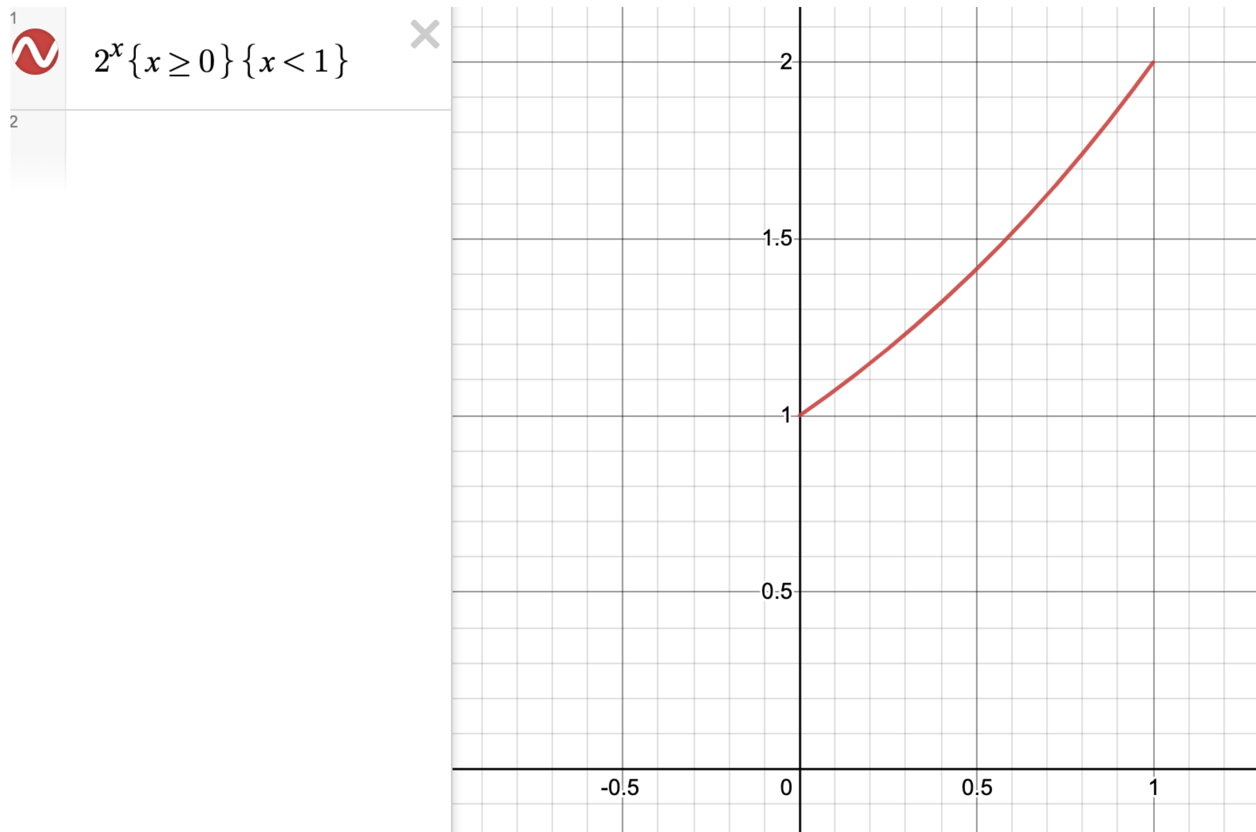


(By Vectorization: Stannered - Own work based on: Float example.PNG, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3357169)

So the C++ code to compute the exponential of the integer part becomes:

```
i = (int)floor(x);
exp2i = bit_cast<float>((i + 127) << 23);
```

Computing the exp_2 of the fractional part of the argument means we only have to deal with values in the $[0, 1)$ interval. In other words, we need to find an approximation for the segment shown below, and make sure the error stays below the corresponding segment in the graph above.



We could resort to using a polynomial which satisfies several arbitrarily chosen conditions, like I've done for the implementation of sine and cosine. Unfortunately, for simple conditions like matching $exp_2(x)$ at equidistant points, such an algebraic solution requires a fairly high degree for the polynomial to have an error of only a few ULP.

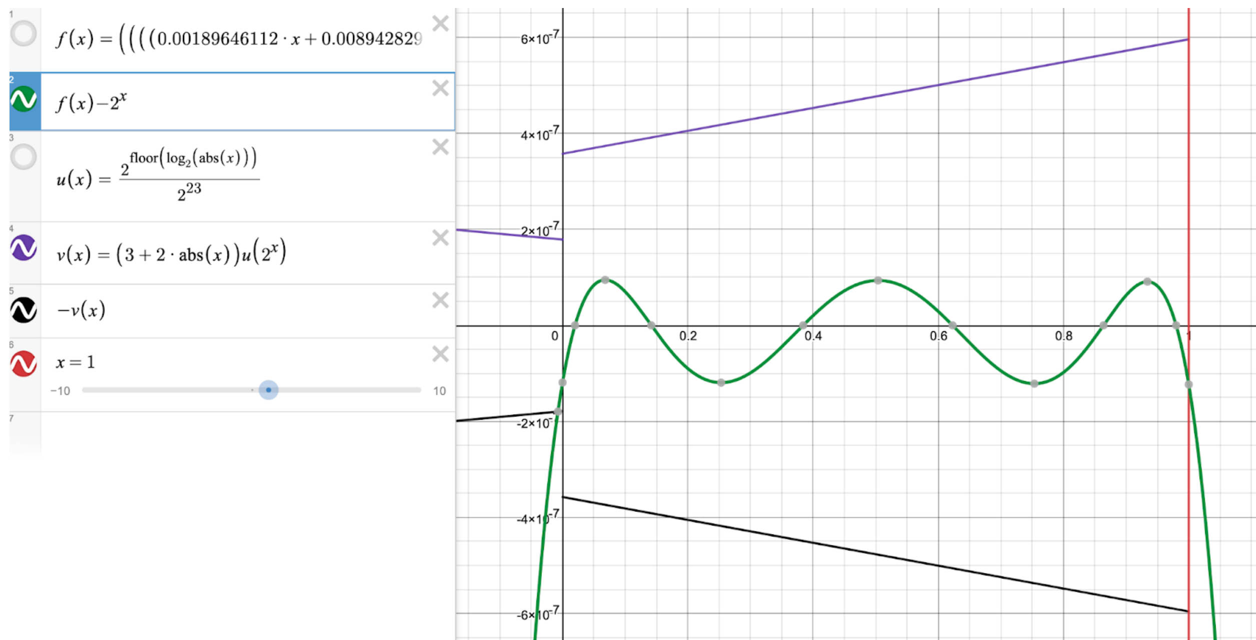
Instead we're better off using an algorithmic solution for the polynomial which best fits this curve. The Remez algorithm computes such a "minimax" polynomial. We'll be using the *LolRemez* program. It's a great open-source implementation of the Remez algorithm with useful features and a simple command-line interface.

We need a 5th degree polynomial to satisfy Vulkan's accuracy requirement:

```
$ ./lolremez --float -d 5 -r "0:1" "2^x"
// Approximation of f(x) = 2^x
// on interval [ 0, 1 ]
// with a polynomial of degree 5.
// p(x)=((((1.8964611e-3*x+8.942829e-3)*x+5.5866246e-2)*x+
```

```
//          2.4013971e-1)*x+6.9315475e-1)*x+9.9999989e-1
float f(float x)
{
    float u = 1.8964611e-3f;
    u = u * x + 8.942829e-3f;
    u = u * x + 5.5866246e-2f;
    u = u * x + 2.4013971e-1f;
    u = u * x + 6.9315475e-1f;
    return u * x + 9.9999989e-1f;
}
```

Plotting the error, we get:



At first look, it may appear the maximum error is just 1.0 ULP!

In reality, it's actually 2.56 ULP when using *binary32* floating-point arithmetic. This is caused by rounding errors introduced in the polynomial's multiplications, and loss of significance in the additions. When we look at the error *relative* to Vulkan's tolerance, which I'll call the 'margin', it peaks at 0.855. Quite a bit worse than what we'd expect from looking at the above pretty graph, but still, any worst case margin value below 1.0 is within Vulkan's requirements. Further improvements on this are just a bonus.

Integer Exactness

One issue with the above solution is that $exp_2(0)$ does not equal 1.0 as expected. Nor do any other integer input values result in exact powers of 2. While Vulkan doesn't spell this out as a requirement, applications expect it, and it appears to be standard across GPU implementations.

The problem with the current polynomial is that the constant coefficient added at the end is not 1.0 but slightly less. We'd really like a polynomial of the form $f(x) \cdot x + 1$, where $f(x)$ is a different Remez-optimized polynomial. Since we want $f(x) \cdot x + 1$ to approximate 2^x, we get:

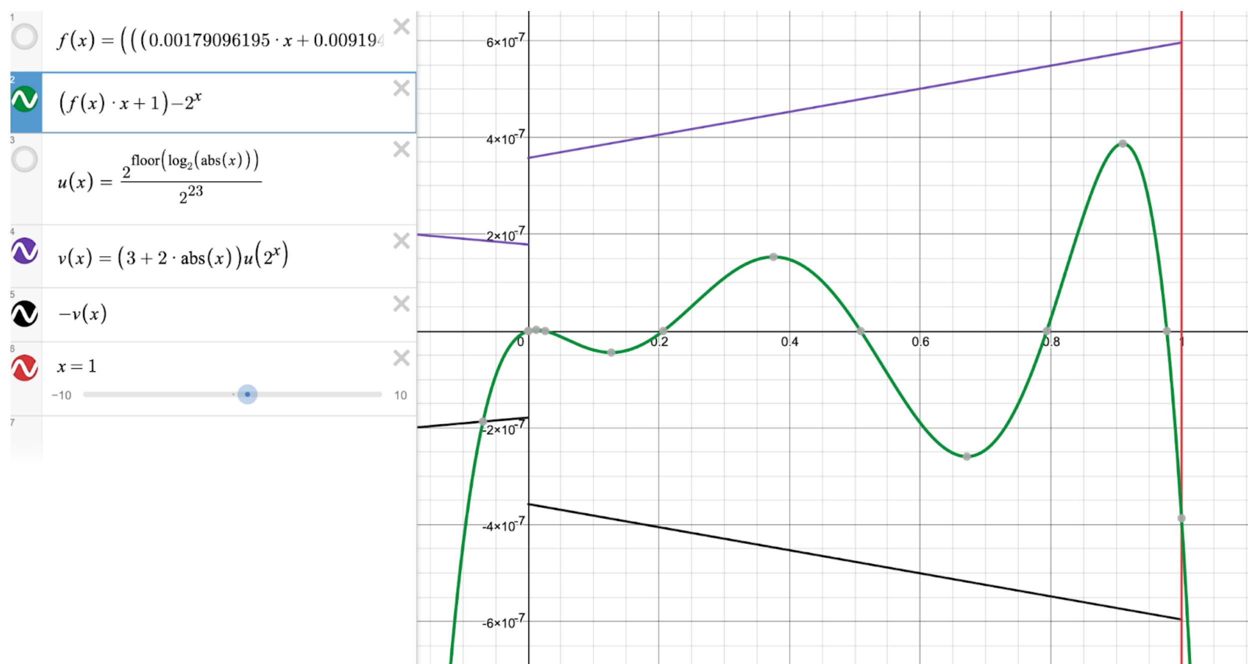
$$f(x) \cdot x + 1 \approx 2^x$$

$$f(x) \approx (2^x - 1) / x$$

In other words, we ask the Remez algorithm to approximate $(2^x - 1) / x$ in the interval $[0, 1]$. Since we're already multiplying $f(x)$ by x, let's see if we can have our cake and eat it by trying to make the new $f(x)$ a 4th degree polynomial:

```
$ ./lolremez --float -d 4 -r "0:1" "(2^x - 1) / x"
// Approximation of f(x) = (2^x - 1) / x
// on interval [ 0, 1 ]
// with a polynomial of degree 4.
// p(x)=(((1.7909619e-3*x+9.1942073e-3)*x+5.5660287e-2)*x+
//       2.4020655e-1)*x+6.9314759e-1
float f(float x)
{
    float u = 1.7909619e-3f;
    u = u * x + 9.1942073e-3f;
    u = u * x + 5.5660287e-2f;
    u = u * x + 2.4020655e-1f;
    return u * x + 6.9314759e-1f;
}
```

Note that the formula this `f(x)` approximates is undefined for $x = 0$, but *LolRemez* still arrives at a solution. Plotting the resulting error:



Looks good enough, right? Unfortunately, no, the margin is exceeded by 1.56.

Interestingly this happens for a very small negative value of x. Despite our algorithm now giving the exact result for $x = 0$, for small negative x values we're computing $exp_2(-1) \cdot exp_2(x+1)$. In other words our polynomial gets evaluated for a value near 1, where its error is the greatest. Meanwhile, left of $x = 0$ the error tolerance is much narrower. It's ~3 ULP, but half as much in absolute value compared to the ULP for small positive x values. While $exp_2(-1) = 0.5$ so the error is also reduced by a factor of 2, it essentially means we need to keep things below 3 ULP across the interval of the polynomial; Vulkan's extra tolerance of an additional $2 \times |x|$ ULP doesn't really help us here.

We could try making the error near $x = 1$ smaller, or address the asymmetry issue. Or both?

Watch your Weight

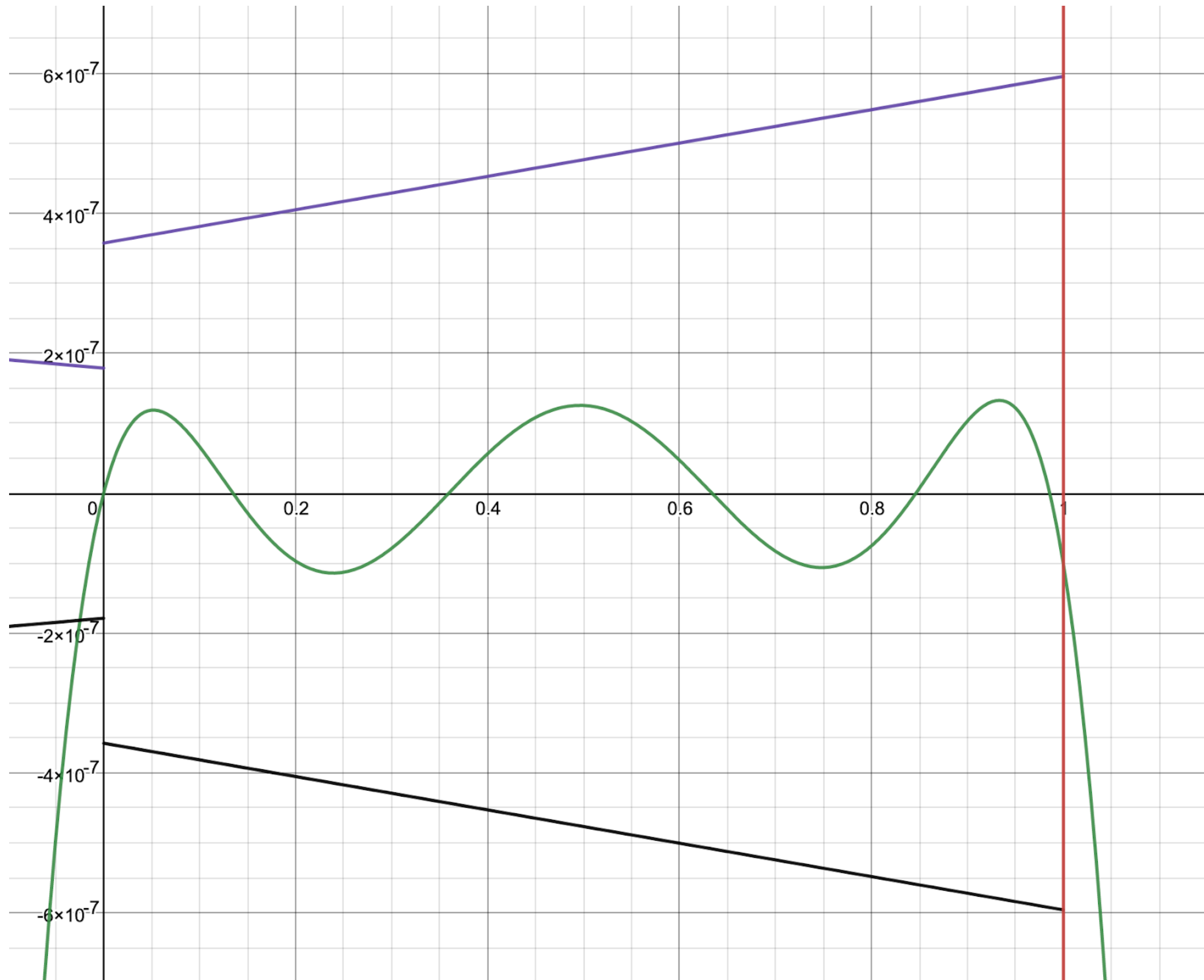
While the basic Remez algorithm always optimizes the polynomial to have the minimum maximum absolute error, i.e. a constant across the interval it is optimizing for, it's relatively easy to "shape" the error envelope by using a weight function.

Note that the *LolRemez* program considers the weight function to define the allowed relative tolerance. A *larger* weight in a certain region means the polynomial will be *less* accurate there. Others consider the weight function to define the inverse of that, which might sound more natural as a "weight", but is less intuitive when trying to match a given tolerance function. Just think of *LolRemez*'s weight function as "tolerance" and it becomes both intuitive and practical.

So what weight function should we use for our `exp2()` approximation? As noted earlier, $ulp(x)$ is actually constant across the $[0,1)$ interval. And the error graph for the 5th degree polynomial approximation of 2^x clearly stays within a band of ~1.0 ULP. But when we started using $f(x) \cdot x + 1$ instead, the error near $x = 0$ became 0 but then appears to get larger with increasing x. That's no coincidence. With $f(x)$ optimized for a constant error, the multiplication by x "squeezes" the error to become zero at $x = 0$ and then linearly grows. To undo that, we need to use $1/x$ as the weight function.

The result looks like this:

```
$ ./lolremez --float -d 4 -r "0:1" "(2^x-1)/x" "1/x"
// Approximation of f(x) = (2^x-1)/x
// with weight function g(x) = 1/x
// on interval [ 0, 1 ]
// with a polynomial of degree 4.
// p(x)=(((1.8852974e-3*x+8.9733787e-3)*x+5.5835927e-2)*x+
//       2.4015281e-1)*x+6.9315247e-1
float f(float x)
{
    float u = 1.8852974e-3f;
    u = u * x + 8.9733787e-3f;
    u = u * x + 5.5835927e-2f;
    u = u * x + 2.4015281e-1f;
    return u * x + 6.9315247e-1f;
}
```



Magnificent! In practice the single-precision results are also good: 2.66 ULP and a margin value of 0.85.

Enjoy that extract-at-integers cake (but keep watching your weight)!

Symmetrical Range Reduction

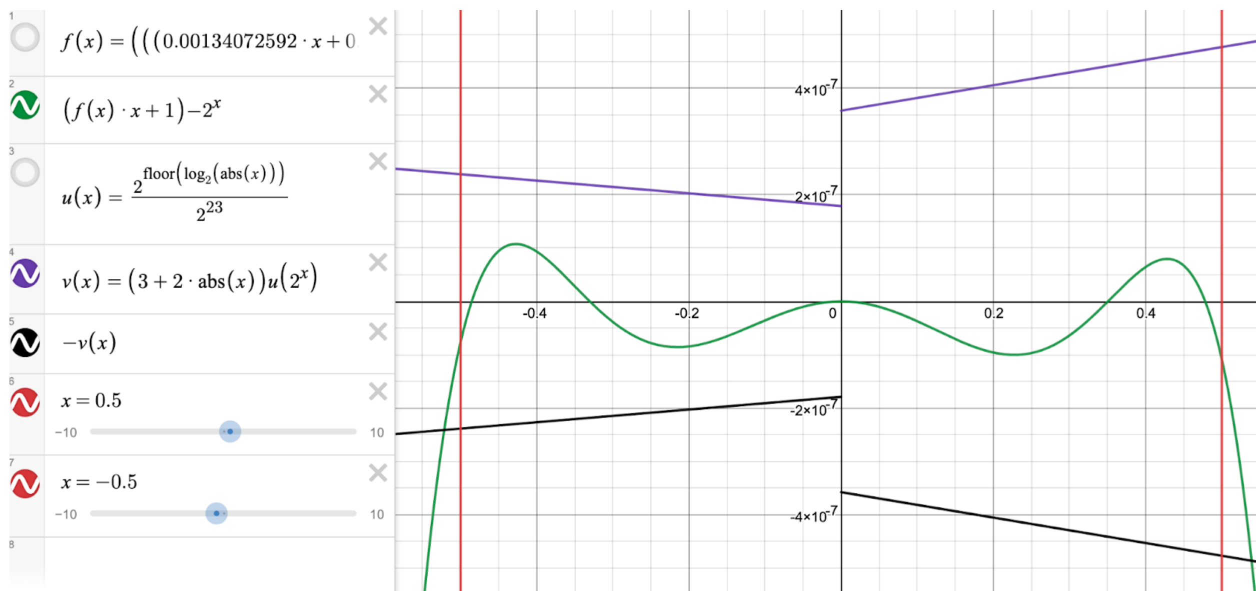
While we can tweak the weight function a bit to achieve slightly better results, recall that the asymmetry between small negative values of x and small positive values forced us to practically ignore the $2 \times |x|$ part of Vulkan's tolerance. Also the numerical losses are greater near x=1 so counter to Vulkan's formula things only improve when making the weight function smaller near that end. Can we avoid having small negative values of x "wrap around" to evaluate the polynomial near 1?

Fortunately, yes. The mathematical identity $exp_2(x_1 + x_2) = exp_2(x_1) \cdot exp_2(x_2)$ allows x to be split into *any* sum of values that add up to x. The pair of $floor(x)$ and $x - floor(x)$ was an obvious

first choice, but it results in skewing the second term to be in the $[0,1)$ interval. We can instead use $round(x)$ and $x - round(x)$, which gives us a $[-0.5, 0.5]$ interval for the latter.

Aside from replacing `floor(x)` with `round(x)`, we just have to update our polynomial with the result *LolRemez* produces when shifting the interval to $[-0.5, 0.5]$:

```
$ ./lolremez --float -d 4 -r "-0.5:0.5" "(2^x-1)/x" "1/x"
// Approximation of f(x) = (2^x-1)/x
// with weight function g(x) = 1/x
// on interval [ -0.5, 0.5 ]
// with a polynomial of degree 4.
// p(x)=(((1.3407259e-3*x+9.6718751e-3)*x+5.5503084e-2)*x+
//        2.4022235e-1)*x+6.9314721e-1
float f(float x)
{
    float u = 1.3407259e-3f;
    u = u * x + 9.6718751e-3f;
    u = u * x + 5.5503084e-2f;
    u = u * x + 2.4022235e-1f;
    return u * x + 6.9314721e-1f;
}
```



This produces an effective ULP error of 2.80, which is slightly worse than our previous best of 2.66 ULP. However, the 'margin' measure improved from 0.85 to 0.73.

But that's far from the best achievable result. The weight function for the above polynomial was still $1/x$, or just a constant 1 when taking the subsequent multiplication by x into account. We

could use e.g $(0.25 \cdot erf(1000 \cdot x) + 0.75) \cdot (3 + 2 \cdot |x|)$ as the numerator instead, to practically match the shape of Vulkan's tolerance function. This gives us 2.29 ULP and a 0.59 margin.

While that weight function should theoretically produce the best result, practice disagrees once again. With a bit of trial and error, I found that a weight function of $(3 + 3.5 \cdot x) / x$ produces an even better result: 1.88 ULP, and 0.54 margin.

Watch your Extremities

There's a subtle problem with computing $exp_2(x)$ as $exp_2(round(x)) \cdot exp_2(x - round(x))$. For example for $x = 127.7$, the first term would overflow to infinity, and while the second term evaluates to 0.648, the end result is still infinity. Meanwhile $exp_2(127) \cdot exp_2(0.7)$ can be computed as a normal *binary32* value just fine.

Many applications won't care about this, but some do. The Vulkan conformance test suite checks the result for these kinds of input values. Note also that $exp(x)$ is typically computed as $exp_2(1/ln(2) \cdot x)$, and $pow(x, y)$ as $exp_2(y \cdot log_2(x))$, which creates several more opportunities for overflow or underflow when the result should still be representable.

Note that we could compute the equivalent $exp_2(round(x) - 1) \cdot (2 \cdot exp_2(x - round(x)))$ at no extra cost and no loss of accuracy, and it avoids intermediate overflow, but it suffers from underflow. Note that Vulkan does not require support for subnormal floating-point numbers, while many CPUs do support them, so this can help avoid prematurely returning 0.0 for large negative input values. Unfortunately computations involving subnormal numbers can incur a performance penalty, and so it's desirable to enable the CPU's flush-to-zero mode.

Alas, this means the round()-based implementation of exp2() is not really suitable for a Vulkan implementation on the CPU, despite its attractive properties.

In any case, we need to clamp the range of the input argument to ensure we overflow to the proper representation of infinity (and not NaN), and underflow to 0 (and not a negative number).

Note that a GPU-based implementation typically obtains polynomial coefficients from a table, and thus optimizes them for different intervals. We could do something similar on the CPU, but clearly it would be much more costly than simply improving precision through a higher degree polynomial.

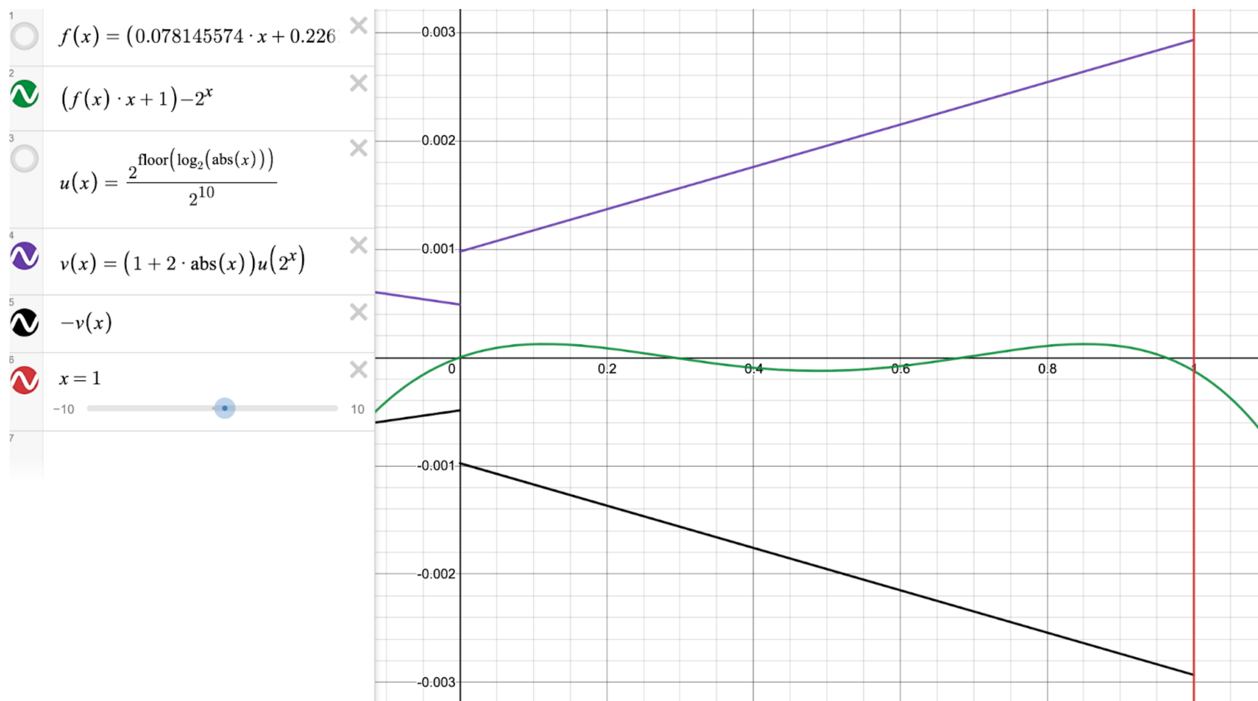
Relaxing with a Halfling

While we've reached the limits of improving accuracy without adding more computational cost, we can reduce the cost by sacrificing accuracy. Fortunately Vulkan and SPIR-V have just the thing to allow for that: Relaxed Precision. GLSL 4.6 supports expressing relaxed precision requirements through the `mediump` qualifier. It essentially reduces the accuracy requirement to that of the half-precision *binary16* format, while still producing 32-bit results. Specifically for

$exp_2(x)$ the tolerance becomes $1 + 2 \times |x|$ ULP. Note this is in ULP-16, which is much coarser than a ULP-32.

Hitting this accuracy requirement is easily feasible with a 3rd degree polynomial (2nd degree when factoring out x to still get exact results at integer values):

```
$ ./lolremez --float -d 2 -r "0:1" "(2^x-1)/x" "1/x"
// Approximation of f(x) = (2^x-1)/x
// with weight function g(x) = 1/x
// on interval [ 0, 1 ]
// with a polynomial of degree 2.
// p(x)=(7.8145574e-2*x+2.2617357e-1)*x+6.9555686e-1
float f(float x)
{
    float u = 7.8145574e-2f;
    u = u * x + 2.2617357e-1f;
    return u * x + 6.9555686e-1f;
}
```



This achieves 0.13 ULP, and could be tuned a bit to trade ULP for an improved 'margin' result, but I kind of like the universality of ULP versus Vulkan's seemingly arbitrarily sloping tolerance function.

So we save two multiplies and two additions, or two fused multiply-add (FMA) operations. It's significant, but we're still left with requiring quite a lot of other instructions:

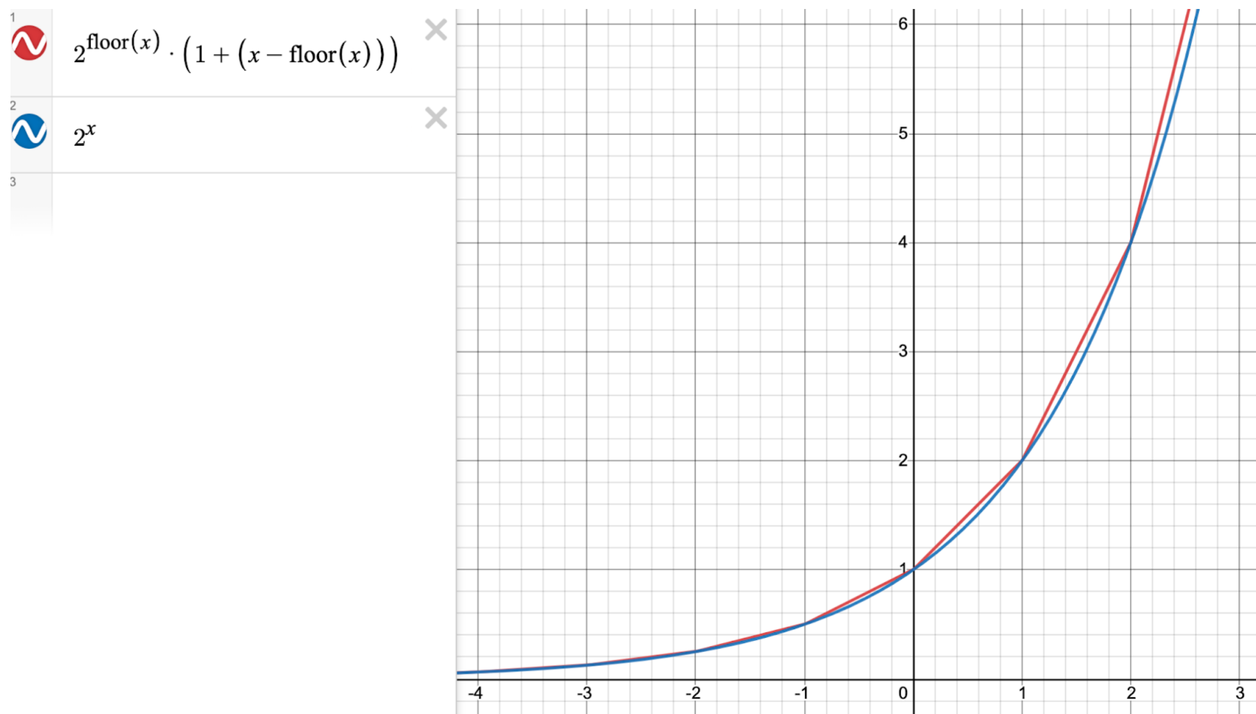
```
float exp2_relaxed(float x)
{
    x = min(x, 128.0f);
    x = max(x, bit_cast<float>(0xC2FDFFFF));  // -126.999992

    float x0 = floor(x);
    int i = (int)x0;
    float f = x - x0;

    // Add single-precision bias, and shift into exponent.
    float exp2i = bit_cast<float>((i + 127) << 23);

    const float a = 7.8145574e-2f;
    const float b = 2.2617357e-1f;
    const float c = 6.9555686e-1f;
    float exp2f = fma(fma(fma(a, f, b), f, c), f, 1.0f);

    return exp2i * exp2f;
}
```

Note I'm only using `fma()` here to illustrate the theoretical cost. The standard C++ function may incur call overhead and may actually be implemented using double-precision operations.

Looking at the line to compute `exp2i`, there's an addition and a shift operation. A shift left is just a multiplication by a power of 2. So let's replace these two instructions with a single FMA operation:

```
float exp2i = bit_cast<float>((int)fma(x0, 1 << 23, 127 << 23));
```

Note how this takes a float as input and returns a float, but the input is expected to be a value without a fractional part. What would happen if it did have a fractional part? The fraction would get multiplied by 2^{23}, resulting in a value always smaller than 2^{23}, then it gets converted to an integer, and ends up in the mantissa field of the result (while the integer part of the input still ends up in the exponent field, with the bias added). This gets interpreted as $2^i \cdot (1 + f)$, meaning the fractional part causes linear interpolation between the powers of 2. It's an approximation of 2^x all in itself!



To compute $2^{i+x_f} = 2^i \cdot 2^{x_f}$ instead, where $i + x_f$ equals our original x, we see that $(1+f) = 2^{x_f}$ and thus $f = 2^{x_f} - 1$ is the new fraction we should be using. This can be achieved either by adding it to $floor(x)$, or by adding $f - x_f$ to x. Note that in either of these cases, f or $f - x_f$ can be efficiently computed as a polynomial. The choice appears arbitrary…

```
float exp2_relaxed(float x)
{
    x = min(x, 128.0f);
    x = max(x, bit_cast<float>(0xC2FDFFFF));  // -126.999992

    float f = x - floor(x);

    // Polynomial which approximates (2^f-f-1)/f.
    const float a = 7.8145574e-2f;
    const float b = 2.2617357e-1f;
    const float c = -3.0444314e-1f;
    float r = fma(fma(a, f, b), f, c);

    // Final multiplication by f, then add to x.
    float y = fma(r, f, x);

    return bit_cast<float>((int)fma(y, 1 << 23, 127 << 23));
}
```

This code not only eliminates the integer addition and shift, it also avoids an extra multiplication, as well as loading the 1.0f constant. The accuracy is still the same as the one above.

Gaining Significance

Wait a sec… the same optimization could also be applied to the high accuracy version, right?

Not really. I've previously mentioned FMA operations only in the context of improving performance. If I replace the `fma()` calls in the `exp2_relaxed()` with multiplies and additions, the achieved accuracy remains the same. But when we use the same algorithm for the high accuracy version (i.e. using the higher degree polynomial), the ULP-32 error increases to 33.4, failing to meet Vulkan's requirement. This result is obtained with both separate multiplies and additions, and with 32-bit FMA operations, like the one below for x86 CPUs supporting [FMA3](#).

```
#include <immintrin.h>

float fma32(float x, float y, float z)
{
    return bit_cast<float>(_mm_extract_ps(_mm_fmadd_ss(
        _mm_set1_ps(x), _mm_set1_ps(y), _mm_set1_ps(z)), 0));
}
```

Using a higher degree polynomial doesn't help either. It's only when the last FMA operation — the one where we multiply by `1 << 23` for the mantissa shift and add `127 << 23` for the exponent bias — is replaced using integer addition, that we get a Vulkan conformant implementation:

```
// lolremez --float -d 4 -r "0:1" "(2^x-x-1)/x" "1/x"
float P(float x)
{
    float u = 1.8852974e-3f;
    u = u * x + 8.9733787e-3f;
    u = u * x + 5.5835927e-2f;
    u = u * x + 2.4015281e-1f;
    return u * x + -3.0684753e-1f;
}

float exp2(float x)
{
    x = min(x, 128.0f);
    x = max(x, bit_cast<float>(0xC2FDFFFF));  // -126.999992
```

```
        float f = x - floor(x);
        float y = P(f) * f + x;

        return bit_cast<float>((int)(y * (1 << 23)) + (127 << 23));
    }
```

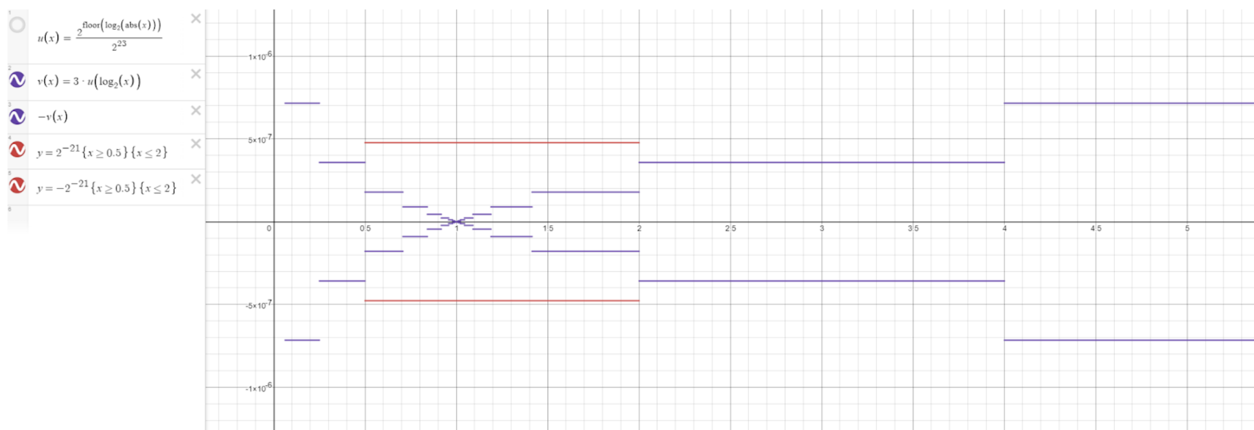
This achieves an excellent accuracy of 2.15 ULP (2.06 when using `fma32()`). The reason behind this gain —the last fully conformant result was 2.66 ULP— is that instead of doing a multiplication by a polynomial approximation for a function which ranges between 1.0 and 2.0, we're *adding* an approximation for a function which ranges between 0.0 and -0.0861. The former is inevitably only accurate to 2^{-23} since only the mantissa value changes in this range, and this results in a 0.5 ULP rounding error.

Note that it also matters a lot now that we're computing $2^f - f - 1$ and adding it to x, instead of computing $2^f - 1$ and adding it to $floor(x)$.

In terms of performance, we've only really eliminated a shift operation and loading one fewer constants. Note that the multiplication by `1 << 23` is still done in floating-point because 32-bit integer SIMD multiplication may get split into multiple µops.

Logarithm = Exponential^{-1}

For computing $log_2(x)$, let's first take a look at Vulkan's precision requirements: "3 ULP outside the range [0.5,2.0]. Absolute error < 2^{-21} inside the range [0.5,2.0]." Plotting this gives the following result:

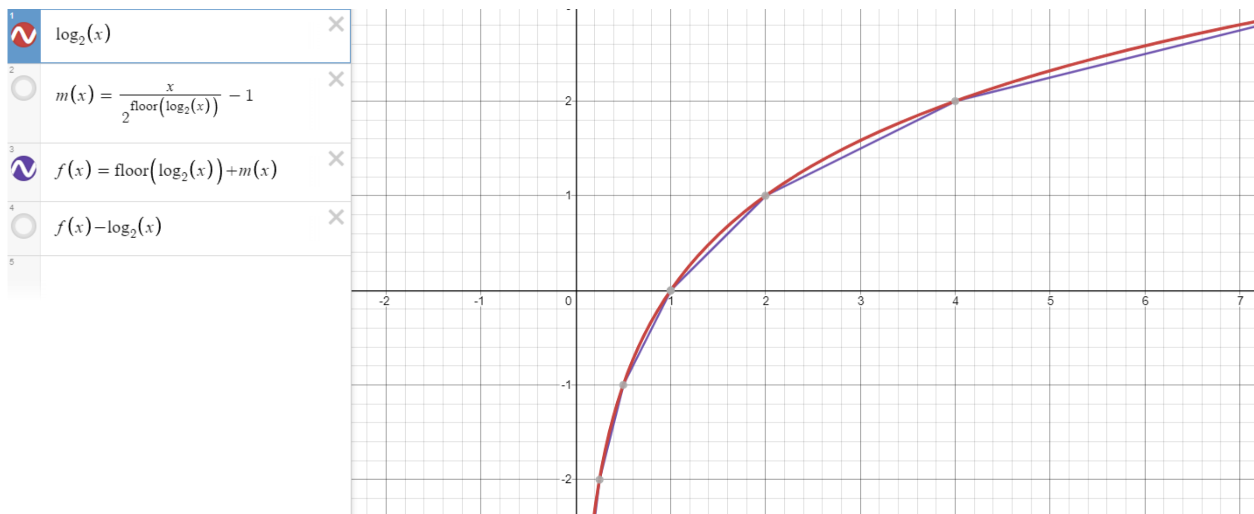


The blue segments are the ULP error, while the red segments are the absolute error. Note that the ULP error tolerance approaches zero near $x = 1$, but it's overridden by the larger absolute error tolerance. And specifically in the interval $[2,4)$ the tolerance is 3 ULP, which translates to an absolute error of $3 \cdot 2^{23}$.

$log_2(x)$ is the inverse of $exp_2(x)$, meaning that $log_2(exp_2(x)) = x$. This also shows in their fundamental identity for range reduction; above we relied on $exp_2(x_1 + x_2) = exp_2(x_1) \cdot exp_2(x_2)$, while with logarithms the roles of addition and multiplication are reversed; $log_2(x_1 \cdot x_2) = log_2(x_1) + log_2(x_2)$.

An efficient implementation can be based on $log_2(2^e \cdot (1 + m)) = log_2(2^e) + log_2(1 + m)$ which simplifies to $e + log_2(1 + m)$, where e is the (unbiased) exponent and m is the mantissa of the input argument. Note again the similarity with $exp_2(x)$: instead of inserting the exponent into the result, for $log_2(x)$ we extract the exponent from the input.

This “extracting” of the exponent can be done by interpreting the input argument as an integer, subtracting `127 << 23`, casting it to floating-point, and multiplying by `1.0f / (2 << 23)`. Sound familiar? It’s exactly the inverse steps of the last line of the `exp2()` implementation above. And unless we call `floor()` on this result to retain just the exponent value, we have a piecewise linear approximation of $log_2(x)$ that is exact at powers of 2:



The next step is inverting `y = P(f) * f + x`. Note that `f` is the fraction of `x`, and for `x` in the interval $[0, 1)$ we can also write `y = P(x) * x + x`. This formula is now an ordinary 5th degree polynomial. Unfortunately, solving that for `x` is the equivalent of finding the roots, and this is not generally solvable, let alone produce a reasonably low-degree polynomial.

Let’s take a step back here. `P(f) * f` was an approximation of $2^f - f - 1$, so the exact formula is $y = exp_2(x) - 1$. Solving that for x we get $log_2(y) + 1$. Remember, this is only correct for the *output* x in the range $[0, 1)$, which corresponds to the *input* y in the interval $[1, 2)$. Recall that y is the result of the piecewise linear approximation, so our actual input interval for which this algorithm is correct is $[2, 4)$. To be exact at powers of 2, namely $x = 2$ and thus $y = 1$, we’re looking for an approximation of the form $P(y) \cdot (y - 1) + 1$. In other words $P(y) \approx log_2(y) / (y - 1)$.

Entering this into *LolRemez*, for various polynomial degrees *D*, we get the following results:

```
$ ./lolremez --float -d D -r "2:4" "log2(x)/(x-1)" "1/(x-1)"
```

	ULP-32 over [2,4)	with FMA	ULP-16
D=1			6.47766
D=2			0.78980
D=3	862.56		0.10535
D=4	124.86		0.015259
D=5	25.176	22.872	0.0030518
D=6	26.249	21.116	0.0031738
D=7	54.680	40.117	

Yikes! For $exp_2(x)$ a polynomial of 4th degree sufficed (5 when including the final multiplication) to get below 3 ULP-32, but for $log_2(x)$ we never get to single-digit ULP errors, not even with FMA operations. Fortunately things look fine for relaxed precision, as an ULP-16 below 1 is achieved with *D*=2, just like for $exp_2(x)$.

Second Try

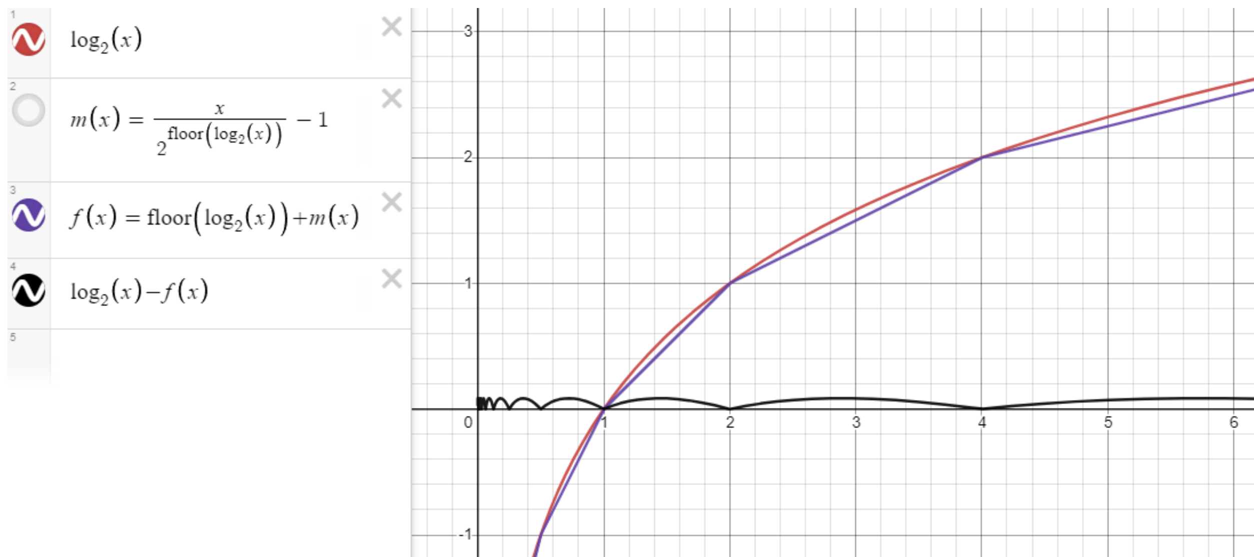
Part of the problem for the high-precision version is that logarithmic functions are fundamentally not that easy to approximate with ordinary polynomials. In fact a Taylor series centered at $x = 1$ never converges past $x = 2$. It's not hard to see that $log_2(x)$ more closely resembles the square root ($x^{0.5}$) or reciprocal function (x^{-1}), so a polynomial with increasing integer powers isn't great at approximating it, even in a limited interval. A second important observation is that the signs of the coefficients of the polynomial approximation alternate, which creates subtractions which cause large losses of significance. For example for degree 5:

```
$ ./lolremez --float -d 5 -r "1:2" "log2(x)/(x-1)" "1/(x-1)"
// Approximation of f(x) = log2(x)/(x-1)
// with weight function g(x) = 1/(x-1)
// on interval [ 1, 2 ]
// with a polynomial of degree 5.
// p(x)=((((-2.645745e-2*x+2.5573874e-1)*x-1.0379186)*x+
//        2.3021687)*x-3.0995312)*x+3.048553
float f(float x)
{
    float u = -2.645745e-2f;
    u = u * x + 2.5573874e-1f;
```

```
        u = u * x + -1.0379186f;
        u = u * x + 2.3021687f;
        u = u * x + -3.0995312f;
        return u * x + 3.048553f;
}
```

This function ranges between 1.436 and 1.0, meaning that only 22 of the mantissa bits play a role, out of the entire 32-bit format. Meanwhile for the computation of $exp_2(x)$ we used a "correction" term which ranges between 0.0 and 0.086, which uses a much larger portion of the *binary32*'s representable range.

Can we do something similar for $log_2(x)$? Here's the difference between the linear approximation and the actual logarithm:



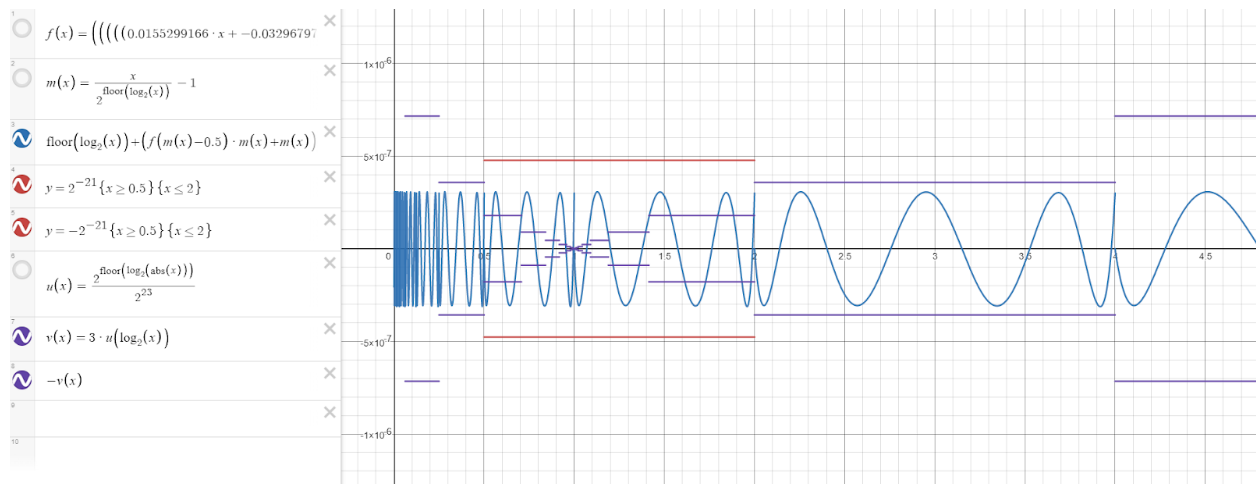
The black graph shows "hops" which are the same shape between each power of 2. This corresponds with the normalized mantissa varying between 0.0 and 1.0. Moreover, this graph only ranges between 0.0 and 0.086, just like $exp_2(x)$'s correction term.

It corresponds to $log_2(m+1) - m$, where m is the mantissa. Once again to get exactness at powers of 2, we separate out a final multiplication and adjust the weight accordingly, and we get:

```
// lolremez --float -d 6 -r "0:1" "(log2(x+1)-x)/x" "1/x"
float P(float x)
{
        float u = 1.5529917e-2f;
        u = u * x + -7.9557731e-2f;
        u = u * x + 1.9429432e-1f;
        u = u * x + -3.2590197e-1f;
```

```
        u = u * x + 4.7355341e-1f;
        u = u * x + -7.2058547e-1f;
        return u * x + 4.4266783e-1f;
}

float log2(float x)
{
    int im = bit_cast<int>(x);
    float y = (float)(im - (127 << 23)) * (1.0f / (1 << 23));

    float m = (float)(im & 0x007FFFFF) * (1.0f / (1 << 23));

    return P(m) * m + y;
}
```

This gives us an effective ULP error of 3.71 in the $[2, 4)$ interval. Much better! But still not below 3, despite already using a 2 degrees higher polynomial than $exp_2(x)$. Let's plot the error to see what's going on:



That's odd… the (blue) error graph stays within the tolerance bounds at all times. It's a tiny margin though. We still suffer loss of significance due to coefficients with alternating signs. All we've done is make the error smaller by making the term itself smaller, by adding it to a linear approximation which itself incurs no precision loss. Even under ideal circumstances, rounding operations introduce errors of 0.5 ULP. The above graph slightly exceeds 2.5 ULP-32 even with Desmos' very high precision arithmetic, meaning we can't achieve less than 3 ULP using this polynomial. FMA operations, weight adjustments, and symmetrical evaluation all help a bit, but can't give us an implementation which satisfies Vulkan's requirements. Bummer.

Fortunately this algorithm behaves well when increasing the degree of the polynomial, and for *D*=7 we get:

```
// lolremez --float -d 7 -r "0:1" "(log2(x+1)-x)/x" "1/x"
float P(float x)
{
    float u = -9.3091638e-3f;
    u = u * x + 5.2059003e-2f;
    u = u * x + -1.3752135e-1f;
    u = u * x + 2.4186478e-1f;
    u = u * x + -3.4730109e-1f;
    u = u * x + 4.786837e-1f;
    u = u * x + -7.2116581e-1f;
    return u * x + 4.4268988e-1f;
}

float log2(float x)
{
    int im = bit_cast<int>(x);
    float y = (float)(im - (127 << 23)) * (1.0f / (1 << 23));

    float m = (float)(im & 0x007FFFFF) * (1.0f / (1 << 23));

    return P(m) * m + y;
}
```

This achieves 1.70 ULP. Success! Weight tuning and FMA operations can lower it a bit further.

The Vanishingly Small and Unfathomably Large

Note that when we extract the mantissa `m`, we immediately divide it by 2^{23} to normalize it to the $[0,1)$ range. While this can be done cheaply by multiplying by the reciprocal, and this doesn't lose any precision, we might consider getting rid of it by "absorbing" it into `P(x)`.

Unfortunately for the high precision log2() this produces a polynomial with coefficients so small they round to zero (it turns out *LolRemez* will happily print numbers that can't be represented in *binary32*, even when using the `--float` argument). It's fine for a relaxed precision version though:

```
// lolremez --float -d 2 -r "0:2^23" "(log2(x/2^23+1)-x/2^23)/x" "1/x"
float f(float x)
{
    float u = 2.8017103e-22f;
    u = u * x + -8.373131e-15f;
```

```
    return u * x + 5.0615534e-8f;
}

float log2_relaxed(float x)
{
    int im = bit_cast<int>(x);
    float y = fma((float)im, (1.0f / (1 << 23)), -127.0f);

    float m = (float)(im & 0x007FFFFF);

    return fma(P(m), m, y);
}
```

Note that it also is able to use `fma()` for the piecewise linear approximation again.

One last thing we need to take care of is the handling of infinity. $log_2(\infty)$ is obviously infinity, but our current implementation returns 128.0, the unbiased exponent of the representation of infinity. We could check for 128.0 and replace it with infinity, but this is not actually correct because `y` becomes 128.0 before `x` becomes infinity, due to rounding. This also happens when using the high-precision version which uses the integer subtraction, because integer values greater than `1 << 23` don't fit in the mantissa (note that this loss of significance isn't otherwise a problem due to 3 ULP being a larger absolute tolerance for larger $log_2(x)$). We can instead deal with infinity like this:

```
float log2(float x)
{
    int im = bit_cast<int>(x);
    float y = (float)(im - (127 << 23)) * (1.0f / (1 << 23));
    if(im == 0x7F800000) y = INFINITY;

    float m = (float)(im & 0x007FFFFF) * (1.0f / (1 << 23));

    return P(m) * m + y;
}
```

Results

All of the above code easily translates into SIMD-friendly implementations. Compared to SwiftShader's legacy implementation of `exp2()`, I've fixed overflow/underflow handling, and I've improved its accuracy from 3.37 ULP (not conformant!) to 2.62 (2.30 with FMA), while reducing instruction count. Compared to the previously used reference implementation it is 22× faster. The relaxed precision implementation is an additional 1.5× faster.

The new `log2()` implementation is 1.5× faster than the legacy implementation, and 28× faster than the reference implementation. The relaxed precision implementation is an additional 1.8× faster.