Approximating the `erfinv` function

Mike Giles*

This chapter presents a new approximation of the `erfinv` function which is significantly more efficient for GPU execution due to the greatly reduced warp divergence.

1 Introduction

Like $\cos x$, $\sin x$, e^x and $\log x$, the error function

$$\texttt{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \, \mathrm{d}t,$$

and its inverse $\texttt{erfinv}(x)$ are a standard part of libraries such as Intel's MKL, AMD's ACML and NVIDIA's CUDA math library.

The inverse error function is a particularly useful function for Monte Carlo applications in computational finance, as the error function is closely related to the Normal cumulative distribution function

$$\Phi(x) \;=\; \frac{1}{\sqrt{2\,\pi}} \int_{-\infty}^x e^{-t^2/2} \, \mathrm{d}t \;=\; \frac{1}{2} + \frac{1}{2} \, \texttt{erf}\left(\frac{x}{\sqrt{2}}\right)$$

so

$$\Phi^{-1}(x) = \sqrt{2} \, \texttt{erfinv}(2x-1).$$

If x is a random number uniformly distributed in the range $(0,1)$, then $y = \Phi^{-1}(x)$ is a Normal random variable, with zero mean and unit variance. Other techniques such as the polar method and Marsaglia's Ziggurat method are usually used to transform pseudo-random uniforms to psuedo-random Normals, but $\Phi^{-1}(x)$ is the preferred approach for quasi-random uniforms generated from Sobol sequences and lattice methods, as it preserves the beneficial properties of these low-discrepancy sequences. To learn more about

*Oxford-Man Institute of Quantitative Finance, Eagle House, Walton Well Road, Oxford OX2 6ED

a) the difference between pseudo-random and quasi-random numbers, b) methods for transforming them from uniform to Normal distributions, and c) their use in financial Monte Carlo simulations, see [Gla04].

Like trigonometric functions, `erfinv`(x) is usually implemented using polynomial or rational approximations [BEJ76, Str68, Wic88]. However, these approximations have been designed to have a low computational cost on traditional CPUs. The single precision algorithm from [BEJ76] which is used in the CUDA math library has the form shown in Table 1.

Table 1: Pseudo-code to compute $y =$ `erfinv`(x) , with $p_n(t)$ representing a polynomial function of t .

$a = |x|$

if $a > 0.9375$ **then**
  $t = \sqrt{log(a)}$
  $y = p_1(t) \,/\, p_2(t)$

else if $a > 0.75$ **then**
  $y = p_3(a) \,/\, p_4(a)$

else
  $y = p_5(a) \,/\, p_6(a)$

end if

if $x < 0$ **then**
  $y = -y$

end if

If the input x is uniformly distributed on $(-1, 1)$, there is a 0.75 probability of executing the third branch in the code, and only a 0.0625 probability of executing the expensive first branch which requires the computation of $\log(a)$ and its square root.

However, on an NVIDIA GPU with 32 threads in a warp, the probability that all of them take the third branch is $0.75^{32} \approx 0.0001$, while the probability that none of them take the first branch is $0.9375^{32} \approx 0.13$. Hence, in most warps there will be at least one thread taking each of the three branches, and so the execution cost will approximately equal the sum of the execution costs of all three branches.

The primary goal of this paper is to improve the execution speed of `erfinv`(x) through constructing single and double precision approximations with greatly reduced warp divergence, i.e. with most warps executing only one main branch in the conditional code. The technique which is used can be easily adapted to other special functions. The MATLAB code which generates the approximations is provided on the accompanying website, along with the CUDA code for the `erfinv` approximations, and test code which demonstrates its speed and accuracy.

The efficiency of the new approximations is demonstrated in comparison with the implementations in CUDA 3.0. While this chapter was being written, CUDA 3.1 was released. Its single precision `erfinv` implementation incorporates the ideas in this chapter, though the code is slightly different to that shown in Table 5. The double precision implementation is still the same as in CUDA 3.0, but a new version is under development.

These approximations were originally developed as part of a commercial maths library providing Numerical Routines for GPUs [NAG09, BTGTW10]. Similar approximations have also been developed independently for the same reasons by Shaw and Brickman [Sha09].

2 New `erfinv` approximations

2.1 Single precision

The new single precision approximation $\texttt{erfinv}_{\text{SP}}$ is defined as

$$\texttt{erfinv}_{\text{SP}}(x) = \begin{cases} x\,p_1(w), & w \le w_1 \qquad \text{central region} \\ x\,p_2(s), & w_1 < w \qquad \text{tail region} \end{cases}$$

where

$$w = -\log(1-x^2), \quad s = \sqrt{w},$$

and $p_1(w)$ and $p_2(s)$ are two polynomial functions.

The motivation for this form of approximation, which in the tail region is similar to one proposed by Strecok [Str68], is that

- `erfinv` is an odd function of x, and has a Taylor series expansion in odd powers of x near $x=0$, which corresponds to $p_1(w)$ having a standard Taylor series at $w=0$;
- `erfinv` is approximately equal to $\pm\sqrt{w}$ near $x=\pm 1$

Using $x = \sqrt{1-e^{-w}}$, the left part of Figure 1 plots $\texttt{erfinv}(x)\,/\,x$ versus w for $0<w<16$ which corresponds to the entire range of single precision floating point numbers x with magnitude less than 1. The right part of Figure 1 plots $\texttt{erfinv}(x)\,/\,x$ versus $s\equiv\sqrt{w}$ for $4<w<36$. The extension up to $w\approx 36$ is required later for double precision inputs.

Using polynomials P_n of degree n, a standard L_∞ approximation for the central region would be defined by

$$p_1 = \arg\min_{p\in P_n} \max_{w\in(0,w_1)} \left| p(w) - \frac{\texttt{erfinv}(x)}{x} \right|.$$

However, what we really want to minimise is the relative error defined as $(\texttt{erfinv}_{\rm SP}(x) - \texttt{erfinv}(x))/\texttt{erfinv}(x)$, so it would be better to define p_1 as

$$p_1 = \arg\min_{p\in P_n} \max_{w\in(0,w_1)} \left| \frac{x}{\texttt{erfinv}(x)} \left(p(w) - \frac{\texttt{erfinv}(x)}{x} \right) \right|.$$

Since this weighted L_∞ minimisation is not possible using MATLAB, p_1 is instead approximated by performing a weighted least-squares minimisation, minimising

$$\int_0^{w_1} \frac{1}{\sqrt{w(w_1-w)}} \left(\frac{x}{\texttt{erfinv}(x)} \left(p(w) - \frac{\texttt{erfinv}(x)}{x} \right) \right)^2 \mathrm{d}w.$$

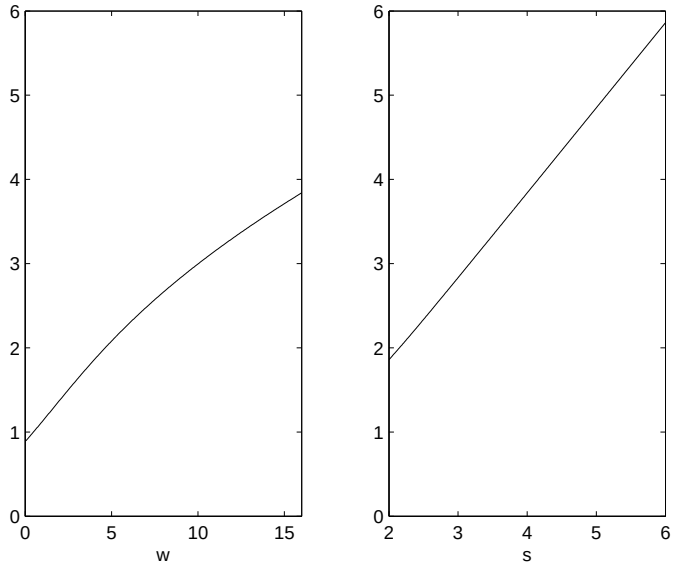

Figure 1: $\texttt{erfinv}(x)\,/\,x$ plotted versus w and $s\equiv\sqrt{w}$.

Table 2: Three alternatives for single precision approximation of `erfinv`

w_1	p_1 degree	p_2 degree	tail prob.
5.00	8	8	0.3%
6.25	9	7	0.1%
16.00	14	n/a	0%

The weighting is a standard one under which Chebyshev polynomials are orthogonal, and is introduced to control the errors near the two ends of the interval. A similar construction is used for p_2 .

Table 1 shows the degree of polynomial required for p_1 and p_2 , to reduce the relative approximation error to less than 10^{-7} , depending on the choice of the dividing point w_1 . The fourth column gives the approximate probability that a random input x , uniformly distributed on $(-1, 1)$, will lie in the tail region. Multiplying this by 32 gives the approximate probability that one or more inputs in a CUDA warp of 32 threads will lie in the tail region, and hence that a CUDA implementation will have a divergent warp.

Using $w_1 = 5$, there is roughly a 10% probability of a divergent warp, but the cost of the central region approximation is the least since it requires only a degree 8 polynomial for p_1 . On the other hand, using $w_1 = 16$, there is no tail region; the central region covers the whole interval. However, p_1 is now of degree 14 so the cost has increased. The second option uses $w_1 = 6.25$, for which p_1 needs to be of degree 9. In this case, there is only a 3% probability of a divergent warp.

2.2 Double precision

The double precision approximation $\texttt{erfinv}_{\text{DP}}$ is defined similarly, but it is defined with up to two tail regions as it needs to extend to $w \approx 36$ to cover the full double precision range for x .

$$\texttt{erfinv}_{\text{DP}}(x) = \begin{cases} x\,p_1(w), & w \le w_1 & \text{central region} \\ x\,p_2(s), & w_1 < w \le w_2 & \text{tail region 1} \\ x\,p_3(s), & w_2 < w & \text{tail region 2} \end{cases}$$

Table 2 shows the degree of polynomial required for p_1 , p_2 and p_3 to

Table 3: Two alternatives for double precision approximation of `erfinv`

w_1	w_2	p_1 degree	p_2 degree	p_3 degree	tail prob.
6.25	16.0	22	18	16	0.1%
6.25	36.0	22	26	n/a	0.1%

reduce the relative approximation error to less than $2{\times}10^{-16}$, depending on the choice of the dividing points w_1 and w_2 . The last column again gives the approximate probability that a uniformly distributed random input x is not in the central region.

In constructing the weighted least-squares approximation using MATLAB, variable precision arithmetic (an extended precision feature of the Symbolic Toolbox) is required to evaluate the analytic error function to better than double precision in order to compute the accuracy of the approximation.

2.3 Floating point error analysis

In this section we look at the errors due to finite precision arithmetic. The floating point evaluation of $(1-x)(1+x)$ will yield a value equal to

$$(1-x)(1+x)(1+\varepsilon_1)$$

where ε_1 corresponds to at most 1 `ulp` (unit of least precision) which is roughly 10^{-7} for single precision and 10^{-16} for double precision. Computing w by evaluating the log of this will yield, approximately,

$$w + \varepsilon_1 + \varepsilon_2$$

where ε_2 is the error in evaluating the log function, which is approximately $5 \times 10^{-8} \max(1, 3w)$ when using the CUDA fast single precision function `__logf()`, and $10^{-16}w$ when using the double precision function `log()`.

Computing $p_1(w)$ will then yield, approximately

$$p_1(w)\left(1 + (\varepsilon_1 + \varepsilon_2)\frac{p_1'(w)}{p_1(w)} + \varepsilon_3\right)$$

where ε_3 is the relative error in evaluating p_1 . The relative error in the

final product $x\,p_1(w)$ will then be approximately

$$(\varepsilon_1 + \varepsilon_2)\frac{p_1'(w)}{p_1(w)} + \varepsilon_3 + \varepsilon_4,$$

where ε_4 is again of order 10^{-7} .

Since $p_1(w) \approx 1 + 0.2w$, the combined contributions due to ε_1 and ε_2 are about 1.5 `ulp` in single precision, and about 0.5 `ulp` in double precision. The errors due to ε_3 and ε_4 will each contribute another 0.5 `ulp`. These are in addition to a relative error of roughly 1 `ulp` due to the approximation of `erfinv`. Hence, the overall error is likely to be up to 4 `ulp` for single precision and up to 3 `ulp` for double precision.

In the tail region, there is an extra step to compute $s = \sqrt{w}$. Since

$$\sqrt{w + \varepsilon} \approx \sqrt{w} + \frac{\varepsilon}{2\sqrt{w}}$$

the value which is computed for s is approximately

$$s + \frac{1}{2s}(\varepsilon_1 + \varepsilon_2) + \varepsilon_5\, s$$

where ε_5 is the relative error in evaluating the square root itself, which for the CUDA implementation is 3 `ulp` in single precision and less than 0.5 `ulp` in double precision. Noting also that $p_2(s) \approx s$, the relative error in the final result is

$$\frac{1}{2w}(\varepsilon_1 + \varepsilon_2) + \varepsilon_5 + \varepsilon_3 + \varepsilon_4,$$

In single precision, ε_2/w is about 3 `ulp` and ε_5 is also 3 `ulp`, so the overall error, including the p_2 approximation error will be about 6 `ulp`. In double precision, ε_2/w and ε_5 are about 1 `ulp` and 0.5 `ulp`, respectively, so the overall error is 2 to 3 `ulp`.

3 Performance and accuracy

Table 4 gives the performance of the first of the new approximations in Tables 2 and 3, compared to the existing ones in CUDA 3.0. Table 5 gives the code for the new single precision approximation. The times are in milliseconds to compute 100M values, using 28 blocks with 512 threads, and each thread computing 7000 values.

There are two conditions for the tests, “uniform” and “constant”. In the uniform case, the inputs x for each warp are spread uniformly in a

Table 4: Times in milliseconds to compute 100M single precision (SP) and double precision (DP) values using CUDA 3.0 on C1060 and C2050

	C1060		C2050	
time (ms)	SP	DP	SP	DP
uniform, old	24	479	31	114
uniform, new	8	219	10	49
constant, old	8	123	11	30
constant, new	8	213	9	48

way which ensures 100% warp divergence for the existing implementation. This worst case scenario is slightly worse than the 87% divergence which would result from random input data, as discussed in the Introduction. The constant case represents the best case scenario in which all of the threads in each warp use the same input value, though this value is varied during the calculation so that overall each conditional branch is exercised for the appropriate fraction of cases.

The results show a factor 3 or more difference in the performance of the existing implementation on the two test cases. This reflects the penalty of warp divergence, with a cost equal to the sum of all branches which are taken by at least one thread, plus the fact that the execution of the main branch (the one which is taken most often) is significantly less costly because it does not require a log calculation.

The cost of the new approximations varies very little in the two cases, because even in the "uniform" case almost all warps are non-divergent. On the other hand, all warps have to perform a log calculation, and therefore in double precision the new implementation is slower than the existing one for the constant case.

Regarding accuracy, the maximum error of the new single precision approximation, compared to the existing double precision version, is around 7×10^{-7} , which is better than the existing single precision version, and the maximum difference between the new and existing double precision implementations is approximately 2×10^{-15} .

Table 5: CUDA code for single precision implementation

```
__inline__ __device__ float MBG_erfinv(float x)
{
  float w, p;

  w = - __logf((1.0f-x)*(1.0f+x));

  if ( w < 5.000000f ) {
    w = w - 2.500000f;
    p =   2.81022636e-08f;
    p =   3.43273939e-07f + p*w;
    p =   -3.5233877e-06f + p*w;
    p =  -4.39150654e-06f + p*w;
    p =    0.00021858087f + p*w;
    p =   -0.00125372503f + p*w;
    p =   -0.00417768164f + p*w;
    p =      0.246640727f + p*w;
    p =       1.50140941f + p*w;
  }
  else {
    w = sqrtf(w) - 3.000000f;
    p =  -0.000200214257f;
    p =   0.000100950558f + p*w;
    p =    0.00134934322f + p*w;
    p =   -0.00367342844f + p*w;
    p =    0.00573950773f + p*w;
    p =    -0.0076224613f + p*w;
    p =    0.00943887047f + p*w;
    p =       1.00167406f + p*w;
    p =       2.83297682f + p*w;
  }

  return p*x;
}
```

4 Conclusions

This chapter illustrates the cost of warp divergence, and the way in which it can sometimes be avoided by re-designing algorithms and approximations which were originally developed for conventional CPUs.

It also illustrates the dilemma which can face library developers. Whether the new double precision approximations are viewed as better than the existing ones depends on how they are likely to be used. For random inputs they are up to 3 times faster, but they can also be slower when the inputs within each warp are all identical, or vary very little.

References

[BEJ76] J.M. Blair, C.A. Edwards, and J.H. Johnson. Rational Chebyshev approximations for the inverse of the error function. *Mathematics of Computation*, 30(136):827–830, 1976.

[BTGTW10] T. Bradley, J. du Toit, M. Giles, R. Tong, and P. Woodhams. Parallelisation techniques for random number generators. *GPU Gems 4, Volume 1*, 2010.

[Gla04] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, New York, 2004.

[NAG09] Numerical Algorithms Group. *Numerical Routines for GPUs.* `http://www.nag.co.uk/numeric/GPUs/`, 2009

[Sha09] W.T. Shaw and N. Brickman. Differential equations for Monte Carlo recycling and a GPU-optimized Normal quantile. Working paper, available from arXiv:0901.0638v3, 2009.

[Str68] A.J. Strecok. On the calculation of the inverse of the error function. *Mathematics of Computation*, 22(101):144–158, 1968.

[Wic88] M.J. Wichura. Algorithm AS 241: the percentage points of the Normal distribution. *Applied Statistics*, 37(3):477–484, 1988.