# DoubleCrunch: Brute-Force Analysis of Worst-Case Errors

Marat Dukhan
Georgia Institute of Technology
College of Computing
School of CSE
mdukhan3@gatech.edu

Richard Vuduc
Georgia Institute of Technology
College of Computing
School of CSE
richie@cc.gatech.edu

## ABSTRACT

Analyzing the accuracy of floating-point implementations of mathematical functions is a hard and complicated problem. The traditional approach to this problem is to use formal specifications and proof-checkers, which employ formal methods to verify the accuracy of an implementation. This paper proposes a very different, complementary approach: use the raw compute power of high-performance co-processors to compute the error for *every* possible double-precision input. The new approach has three important advantages: it fully automates error analysis of floating-point expressions, it can easily handle non-traditional floating-point operations, such as approximate logarithm, and it finds exact error bounds. We describe our brute-force error analysis framework, called DoubleCrunch, implemented on top of CUDA, OpenCL, and SIMD instrinsics. The empirical results suggest that despite the enormous search space, practical problems can be analyzed even on a single workstation; and complete functions could be analyzed in a relatively short time using a supercomputer.

> When in doubt, use brute force.
>
> Ken Thompson [13]

## 1. INTRODUCTION

We consider the problem of how to compute the worst-case error of a double-precision floating-point computation in a single input variable. Examples include computing reciprocals ($1/x$), evaluating polynomials, and evaluating elementary functions (e.g., $\log x$, $\sin x$, $\frac{1}{\sqrt{x}}$), to name a few. We specifically assume the existence of a highly-accurate, but possibly slow, reference implementation; and we wish to check an alternative—and presumably highly-tuned or optimized—implementation against this reference.

While there are a number of powerful symbolic tools and algorithms for rigorously bounding such errors [6, 1, 3], they have found limited use in practice, owing primarily to the difficulty of applying them. Indeed, most widely used library implementations of mathematical functions, such as LibM, do *not* provide such mathematically-guaranteed error bounds, which might otherwise have been derived from such tools; rather, their documented accuracies tend to be based on selective testing on a random subset of inputs. For instance, the CUDA Programming Guide provides accuracy data for its mathematical functions, but with a notice that, "*the error bounds are generated from extensive but not exhaustive tests, so they are not guaranteed bounds*" [4].

Proving rigorous error bounds is hard because there may be multiple sources of errors: approximation errors in polynomials and polynomial fractions; roundoff errors in the evaluation of floating-point expressions; and inaccuracies in tabulated values, among other reasons. Furthermore, these errors might interact in a non-trivial way. For instance, roundoff errors in Newton-Raphson iterations, which in principle converge mathematically, might not actually converge to the desired accuracy in a practical floating-point implementation.

Moreover, formal verification methods have some fundamental shortcomings:

1. Formal verification tools can not fully automate error analysis. The authors of Gappa, a state-of-the-art tool for formal analysis of floating-point expressions, note that, "*Gappa has evolved to include more and more automatic hints, but most real-world proofs still require writing complex, problem-specific hints*" [6]. The need for expert hints is not a drawback of a specific implementation, but rather a fundamental limitation, which arises from the infinite number of ways to rewrite a symbolic expression which needs to be bounded.

2. Formal specifications have different syntax than implementation code, but the two must closely match each other. Even a small difference between the implementation and the model, such as reordering of additions, may invalidate the error analysis. Accordingly, every change to the implementation code must be matched by a similar change to the formal specification, which is time-consuming and error-prone.

3. An increasing trend in modern instruction sets is to include approximate floating-point operations. For instance, Intel SSE, ARM NEON, and IBM QPX all provide instructions for approximate reciprocal and reciprocal square root computation; and IBM's VSX and Intel's MIC and AVX-512 define approximate base-2


*SuperComputing* 2015 Austin, Texas USA

logarithm and exponential instructions. Approximate instructions operate on the bit-level, mapping specific bits of inputs to specific bits of outputs. This behavior is hard to model with relative error bounds and hard to analyze with formal specifications. However, it is easy to emulate on co-processors and to embed into brute-force analysis.

4. Formal verification tools do not find exact maximum errors, but rather upper-bounds on the maximum errors. Due to conservative analysis of interactions between different kinds of errors, they can substantially overestimate the maximum errors. Such overestimations may prompt a numerical analyst to use more computations (e.g., higher-degree polynomials or larger number of Newton-Raphson iterations) than factually needed and cost performance.

In this paper, we propose an approach that complements formal verification: to determine floating-point errors of univariate exprsesions, we leverage the raw compute throughput of modern manycore co-processors to implement a *brute-force* testing scheme of every floating-point number in the input domain.

Intuitively, this approach might sound naïve or might otherwise seem highly impractical, especially for double-precision computations where, for instance, there are $2^{52}$ values in the semi-closed interval, $[1.0, 2.0)$. However, the history of floating-point research has several examples to the contrary. In a 4-year long effort, Lefevre and Muller used a hundred of SPARC workstations to find the worst cases of several correctly-rounded floating-point functions [10]. To deal with the enourmous search space, they they used smart filtering of floating-point numbers, several high-level optimizations, and implemented the most computationally demanding part in SPARC assembly. Recently Fortin, *et al.*, demonstrated manyfold speedups on GPU over many-core CPUs on searching hard-to-round cases and generating polynomial approximations [7]. Encouraged by such work, this paper suggests a framework for analyzing double-precision floating-point mathematical functions. Unlike the efforts of Lefevre and Muller and Fortin et al., our framework targets functions with errors above 0.5 units in the last place (ULP), that is, not perfectly accurate implementations. We offer practical demonstrations of brute-force error analysis using three recent co-processors: the NVIDIA Tesla K40m GPU, the AMD Radeon R280X GPU, the Intel Xeon Phi SE10P.

We emphasize that the purpose of a brute-force approach is *not* to replace the analytical methods of error analysis; rather, it aims to enable a practical and effective *separation of concerns*: accuracy-focused researchers can focus on developing highly-accurate reference implementations of mathematical functions with guaranteed small error bounds, while performance-focused researchers and implementers can have a new automatic tool for comparing their implementations against these reference designs, to find worst-case errors.

**Contributions.** Exhaustive testing in the style of DoubleCrunch is a novel and highly unconventional application of high-performance computing capabilities.

From a technical perspective, we believe DoubleCrunch makes the following contributions:

1. We introduce and describe DoubleCrunch, a software framework for brute-force analysis of worst-case errors for univariate double-precision floating-point expressions. DoubleCrunch fully automates worst-case error analysis and can use CUDA GPUs, OpenCL GPUs, Intel Xeon Phi boards, or FMA-capable CPUs for acceleration. Through our tests we ensured that all supported platforms generate bitwise identical results.

2. Using DoubleCrunch, we analyze the maximum error in double-precision reciprocal computation through Newton-Raphson iterations without fused multiply-add (FMA) operations. We do so by sieving $2^{52}$ double-precision inputs in $[1, 2)$ range. We find that the final error is not only larger than in Newton-Raphson iterations with an FMA, but also crucially depends on the initial approximation.

3. We use DoubleCrunch to compute the maximum roundoff error in 21-degree polynomial evaluation that approximates logarithms on $\left[\frac{\sqrt{2}}{2}, \sqrt{2}\right]$. We demonstrate that the roundoff error computed by DoubleCrunch is smaller than the error bound produced by Gappa, a state-of-the-art tool for floating-point roundoff error analysis.

4. We estimate the time to perform a brute-force error analysis of a *complete* function. This estimate suggests that a univariate floating-point function can be analyzed within just a few days on a modern supercomputer. Our estimate is based on projecting DoubleCrunch performance and efficiency on two error analysis tasks: Newton-Raphson iterations for reciprocal and polynomial evaluation.

**Limitations.** DoubleCrunch is not a universal tool, and we acknowledge its limitations:

1. DoubleCrunch requires a reference high-precision implementation of the floating-point expression or function being analyzed. Such implementations must have provably low error and cannot be implemented without expertise in error analysis. However, once an expert design such reference implementation, DoubleCrunch users can treat it as a black box.

2. DoubleCrunch does not support brute-force analysis of high-precision (quad-precision or double-double) functions or multi-argument double-precision functions. We do not believe that such analysis would be feasible in the near future.

3. Brute-force analysis is computationally demanding, and can take more time than analytical methods of error analysis. For instance, DoubleCrunch computed the roundoff error of polynomial evaluation in Sec. 5 in 58.7 hours on nVidia Tesla K40, while Gappa produces an error bound in 44 miliseconds on Intel Core i7-4770K.

Despite these limitations, we believe DoubleCrunch is a compelling demonstration of HPC to problems in experimental applied mathematics, in the spirit of other proposals [2].

## 2. HIGH-PRECISION ARITHMETIC ON GPUS AND ACCELERATORS

Table 1 summarizes our experimental platforms.

**Table 1: Evaluated compute devices**

| Name (processing units) | DP GFLOPS | Programming model |
|---|---|---|
| nVidia Tesla K40m (15 @ 745 MHz) | 1430 | CUDA or OpenCL |
| Intel Xeon Phi SE10P (61 @ 1100 MHz) | 1074 | MIC intrinsics and OpenMP |
| AMD Radeon R9 280X (32 @ 1020 MHz) | 1044 | OpenCL |
| Intel Xeon E3-1275 v3 (4 @ 3500 MHz) | 224 | AVX2 intrinsics and OpenMP |

## 3. DOUBLECRUNCH FRAMEWORK

The DoubleCrunch framework assists in brute-force search for worst-case errors. The framework has multiple versions (for CUDA, OpenCL, and MIC accelerator devices and for FMA3, FMA4, AVX2, and AVX512-capable CPUs) which share a common command-line driver and store data in the same format. A user provides two functions: `compute_reference` and `compute_approximation`. The `compute_reference` function must calculate a highly accurate reference value of the target function and return it in double-double format. The value does not need to be accurate to full double-double precision; for practical purposes it is enough to provide 10 additional bits of accuracy beyond double precision. The `compute_approximation` function is the implementation with an unknown maximum error to be determined.

To assist the user with implementing `compute_reference` and `compute_approximation`, DoubleCrunch provides a universal set of data types and utility functions that are then mapped to CUDA, OpenCL, or C technologies. Depending on the target platform, the built-in data types DCdouble, DCdoubledouble, DCfloat and DCmask either to scalar C types, or to AVX, MIC or OpenCL vectors. Built-in functions that operate on these data types are provided for polynomial evaluation, error-free transformations, double-double arithmetic, and emulation of special hardware instructions, such as approximate reciprocal instructions on x86 and ARM. If a user implements both `compute_reference` and `compute_approximation` using only built-in functions and data types, DoubleCrunch can offload computations to CUDA, OpenCL, MIC, or many-core CPUs without modifications.

During the brute-force search, DoubleCrunch calls `compute_reference` and `compute_approximation` for each double-precision number in the search range, computes the absolute difference between the reference value and the approximation, and scales it by ULP of the reference value. It then computes the maximum ULP error in a block of $2^{32}$ values and, once a group of such blocks finishes computation on an accelerator, forwards it to the host system and writes to a file. As the search might take many hours and days, DoubleCrunch supports checkpointing; if the search is interrupted, it can restart from the last computed group of blocks. The framework provides a utility, `crunch-status`, to read the DoubleCrunch checkpoint files, even while DoubleCrunch is working. Another utility `crunch-format` aggregates maximum errors of groups of blocks and outputs the aggregated errors in tab-delimited format.

The `compute_reference` and `compute_approximation` functions are inlined into higher-level simulation entry point and compiled by C99, CUDA, or OpenCL compiler. As floating-point codes can be sensitive even to the smallest perturbations, DoubleCrunch configures compilation process to avoid unsafe floating-point optimizations:

1. When targeting CUDA, MIC instrinsics, or AVX-512, DoubleCrunch implements all basic operations on DCdouble using directed rounding intrinsics, e.g. __dadd_rn and _mm512_add_round_pd.
2. On OpenCL DoubleCrunch adds FP_CONTRACT pragma to disable contraction of floating-point expressions.
3. For CPU targets DoubleCrunch adds compilation flags that enforce strict floating-point environment and disables contraction of floating-point expressions.

Through rigirous testing we ensured that the above settings were sufficient to produce bitwise identical results on all supported platforms.

## 4. APPLICATION TO RECIPROCAL COMPUTATION

Many modern instruction sets have instructions for computing approximate reciprocals. For instance, on x86 architectures, the SSE instruction set introduced `RCPSS` and `RCPPS` instructions, which compute approximate reciprocals for a single-precision scalar or SIMD vector. The x86-based Xeon Phi architecture, which does not support SSE, includes the `VRCP23PS` instruction to compute approximate reciprocals for a SIMD vector. Its analogue on the ARM NEON instruction set is called `VRECPE.F32`. These instructions have higher throughput and lower latency than IEEE 754-compliant division, but they pay for enhanced performance with reduced accuracy.

**Table 2: Accuracy of approximate reciprocal instructions**

| Processor family | Max error, SP ULP | Lookup Table (bits in → out |
|---|---|---|
| Intel x86/SSE | 4,995.550 | 11 → 12 |
| Intel x86/MIC | 0.900 | 23 → 23 |
| AMD x86/3dnow! | 532.262 | 15 → 16 |
| AMD x86/SSE (family 15h) | 532.262 | 12 → 12 |
| AMD x86/SSE (other CPUs) | 4,762.276 | 15 → 16 |
| ARM/NEON | 45,502.376 | 8 → 8 |

The accuracy of these instructions is not well-documented, varying on different architectures and even on different families of x86 microarchitecture. However, as the approximate reciprocal instructions operate on single-precision values, it is easy to dump and analyze all possible inputs and outputs.

Table 2 demonstrates the results of our analysis of approximate reciprocal implementations. We found that Intel

Core 2, Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Bonnell, and Silvermont microarchitectures share the same implementation of the SSE approximate reciprocal instruction. Intel Knights Corner (Xeon Phi) has a significantly more accurate approximate reciprocal instruction, with a maximum error below 1 ULP. AMD processors introduced an approximate reciprocal instruction with the 3dnow! instruction set. This instruction was accurate to almost 15 bits, but operated only on a scalar. Later AMD processors supported SSE instructions, which could compute approximate reciprocals on both scalars and SIMD vectors, but their accuracy was reduced compared to their 3dnow! counterparts. The recent AMD Family 15h processors dropped support for the 3dnow! instruction set, but use its more accurate implementation for RCPSS and RCPPS SSE instructions. On all tested x86 processors that support the AVX instruction set, we found no differences between SSE and AVX approximate reciprocal instructions. ARM processors introduced an approximate reciprocal instruction with the NEON SIMD instruction set. We tested the ARM NEON approximate reciprocal instruction on ARM Cortex-A8, Cortex-A9, Cortex-A15, and Qualcomm Krait, and, unlike on x86 processors, found no differences across various vendors and processor families.

**Figure 1: Schema of operation of approximate reciprocal instructions**

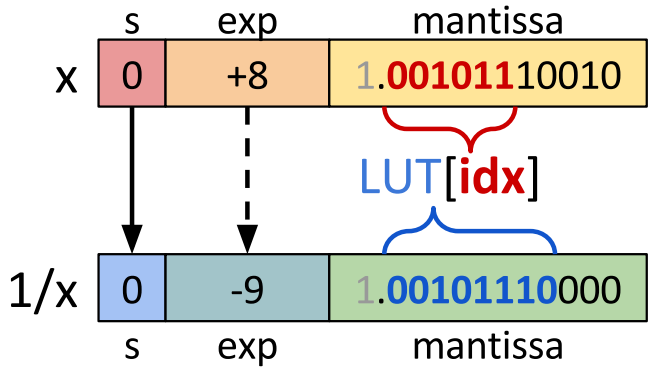
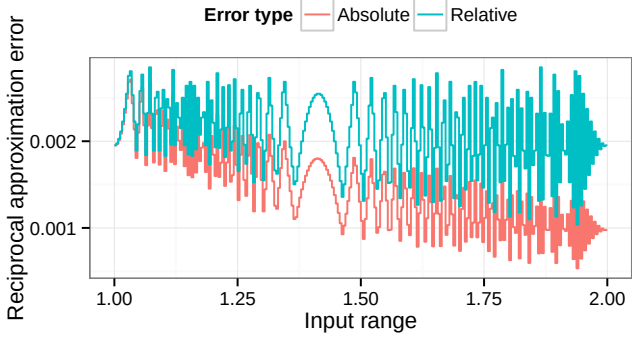



Figure 1 explains how DoubleCrunch emulates approximate reciprocal instructions. The three components of a floating-point number—the sign bit, exponent, and mantissa—are processed independently. The sign bit is copied exactly. The exponent is negated and decremented by 1.[1] However, only few bits of mantissa participate in the transformation of $x \to \frac{1}{x}$: the highest several bits of mantissa of $x$ are used as an index into a table that contains the high bits of the mantissa of $\frac{1}{x}$. Importantly, the low bits of $x$ are ignored by the approximate reciprocal operation, and the low bits of $\frac{1}{x}$ are initialized to zeros. By analysing dumps of inputs and outputs of approximate reciprocal instructions, we found that most such instructions can be emulated with rather small tables. The numbers of input and output bits for the table lookup are specified in Tab. 2. Note that although this model is convenient for software emulation, it is not representative of the actual hardware implementation. The patterns of approximation errors suggest that the tabulated values are not chosen only to minimize the errors, and have additional structure at bit level. A plot of approximation errors in ARM NEON's VRECIP.F32 instruction is presented in Fig. 2

[1]Decrement is needed because when exponent part of x is 0 ($1 < x < 2$), the exponent part of $1/x$ is $-1$ ($\frac{1}{2} < x < 1$)

**Figure 2: Accuracy of VRECIP.F32 approximate reciprocal instruction.**




The accuracy of the approximate reciprocal obtained with a special instruction can be improved by computing a few Newton-Raphson iterations. Performance-wise, it makes sense because most CPUs have non-pipelined floating-point division units, but pipelined multiplication and addition units, and Newton-Raphson iterations need only the latter operations. Mathematically, Newton-Raphson iterations converge to the true value of the reciprocal. However, in an actual floating-point implementation, the roundoff errors might prevent convergence to the correctly rounded value of the reciprocal. When Newton-Raphson iterations are computed with fused multiply-add operations, they provably converge to the correctly rounded value of the reciprocal, except in one special case when the input is 1 ULP less than a power of 2 and the initial approximation is smaller in absolute value than the true value of the reciprocal [11]. When fused multiply-add operations are not available, convergence is not guaranteed, and the maximum error of the approximation after several steps is an open question.

```
DOUBLE_CRUNCH_INLINE
DCdouble compute_approximation(DCdouble x) {
  DCfloat float_x = fcvtd(x);
  // Initial approximation mimicks ARM NEON
  DCfloat y_init = approx_recip_arm_neon(float_x);
  DCdouble y0 = dcvtf(y_init);
  // Newton-Raphson iterations
  DCdouble y1 = dmul(y0, dsub(dconst(2.0), dmul(y0, x)));
  DCdouble y2 = dmul(y1, dsub(dconst(2.0), dmul(y1, x)));
  DCdouble y3 = dmul(y2, dsub(dconst(2.0), dmul(y2, x)));
  return y3;
}

DOUBLE_CRUNCH_INLINE
DCdoubledouble compute_reference(DCdouble x) {
  // Use DoubleCrunch built-in function
  return ddrcpd(x);
}
```

We used DoubleCrunch to resolve this question on our evaluation platforms. Specifically, we simulate 3 Newton-Raphson iterations for reciprocal computation using the initial approximations produced by Intel SSE and ARM NEON implementations. The simulation considers all double-precision

numbers in the half-closed interval, $[1.0, 2.0)$.[2] Each double-precision number $x$ is converted to single-precision with round-to-nearest-even rounding. Then we simulate the x86 `RCPPS` or ARM `VRECIP.F32` instructions on the single-precision input, extend the result to double-precision and denote it $y_0$. Then, we perform three Newton-Raphson iterations:

$$y_{n+1} = y_n \cdot (2 - y_n \cdot x)$$

where $y_n$ is the reciprocal approximation after $n$ Newton-Raphson iterations. At the end, we compute the absolute difference between $y_3$ and the value of $1/x$ computed in double-double arithmetic and convert the absolute error to ULPs of $1/x$. The results of this simulation appear in Table 3 and Fig. 3.

**Table 3: Accuracy of double-precision approximate reciprocal refined with 3 Newton-Raphson iterations without fused multiply-add**

| Initial approximation | Max error, double precision ULP |
|---|---|
| RCPPS on Intel x86/SSE | 1.98309 |
| VRECIP.F32 on ARM/NEON | 1.99999 |

**Figure 3: Accuracy of Newton-Raphson reciprocal computation with initial approximation by `VRECIP.F32`**

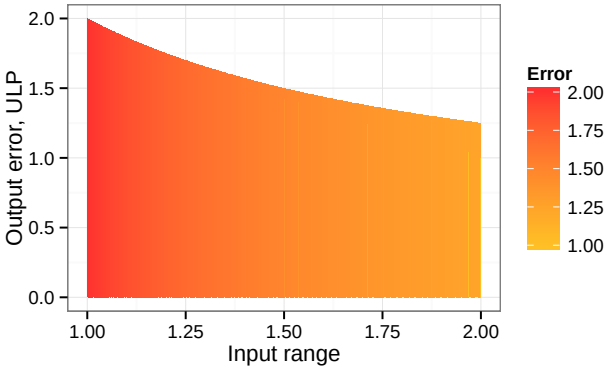
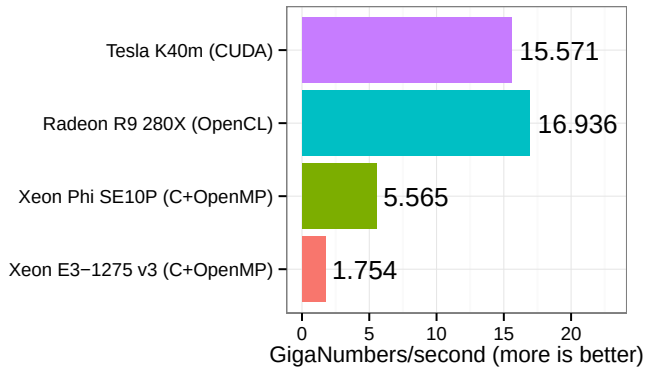



Fig. 4 illustrates the performance of this DoubleCrunch simulation on the three accelerators. Although Tesla has more compute power than Radeon, the latter performs better in this simulation: Tesla has 37% more FLOPS, yet Radeon delivers 9% times better performance. We believe that the extra performance per FLOPS on the Radeon is due to a particularly efficient implementation of division: on the AMD GCN architecture, double-precision division requires only 2 instructions, while on nVidia Kepler it decomposes into multiple Newton-Raphson iterations. Two reasons explain why Xeon Phi co-processor underperforms on this simulation. First, on Xeon Phi the double-precision vector division operation is not supported in hardware; instead, it is implemented with multiple simpler instructions. Secondly, on Xeon Phi any integer, boolean, single-precision, gather, or mask manipulation operations occupy instruction slots that could be used for double-precision operations, thus decreasing the effective double-precision peak.

[2]All non-zero finite floating-point numbers can be scaled into this interval and the scale factor can be inverted at low cost.

**Figure 4: Performance of Newton-Raphson Reciprocal Simulation**



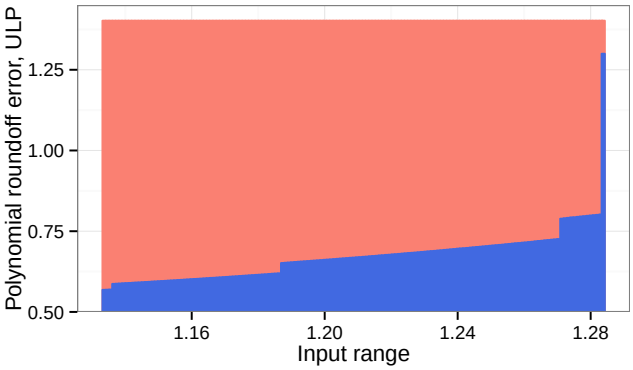

## 5. APPLICATION TO POLYNOMIAL EVALUATION

Newton-Raphson reciprocal simulations exhibits patterns that are inefficient on accelerators. To get a data point on the other side of efficiency spectrum we simulated the evaluation of a 21-degree polynomial approximation to the function `log` $x$ on $\frac{\sqrt{2}}{2} \le x \le \sqrt{2}$. The polynomial has a special form with $c_0 = 0$ and $c_1 = 1$ and is evaluated as:

$$\log x \approx t + t \cdot (t \cdot P_{19}(t))$$

where $t \equiv x - 1$ and $P_{19}$ is a general-form 19-degree polynomial evaluated through a Horner scheme with fused multiply-add.

**Figure 5: Roundoff Errors in Polynomial Evaluation**



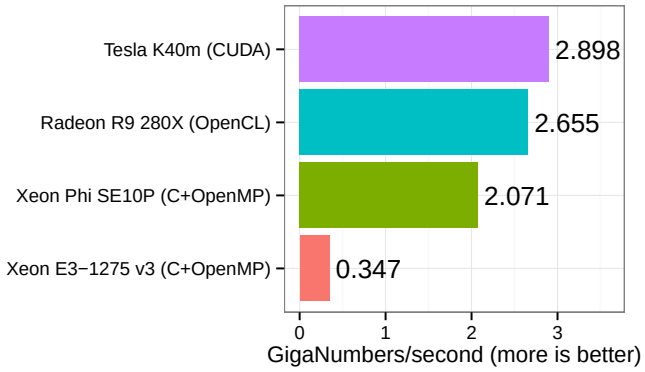

To get reference values, we evaluate the same polynomial with compensated Horner scheme[8], which is equivalent to evaluation of the polynomial in double-double arithmetic, but requires fewer operations. Our implementation of compensated Horner scheme mostly follows[9], but instead of adding the coefficients of error polynomials and evaluating one polynomial, we evaluate both error polynomials and then accumulate the result. The scheme with addition of polynomials saves floating-point operations by trading

FMAs for additions; however, on nVidia Kepler and Intel Knights Corner architectures the cost of FMAs and additions is the same[3], but computing two polynomials in parallel improves accuracy and exposes more instruction-level parallelism.

The roundoff errors found by DoubleCrunch are substantially lower than the upper bounds estimated by Gappa, a state-of-the-art tool for floating-point error analysis. The structure of the errors is presented on Fig. 5 in blue, and Gappa estimate is shown in red.

Performance results for this simulation are depictured in Fig. 6. On this task the CPU is disproportionally slower than accelerators. We attribute this effect to imperfect latency hiding in long dependency chains inside the compensated Horner algorithm.

**Figure 6: Performance of Polynomial Evaluation Simulation**




## 6. PERFORMANCE PROJECTION

While DoubleCrunch can be used on a single machine to search a subset of double-precision numbers, an interesting question is whether it is feasible to do brute-force error analysis for a complete LibM function on its whole domain. In this section we attempt to answer this question without running an actual large-scale simulation.

To analyse a complete mathematical function DoubleCrunch needs its high-precision version that would be used as a reference. Fortunately, the CRLibM[5] library internally provides exp function accurate to 113 bits[4] as a non-public implementation detail. To analyse feasibility of brute-force analysis, we ported this implementation to CUDA and used as a reference high-accuracy implementation of exp function in DoubleCrunch. We choose exp implementation from the popular Cephes library[12] as an approximate implementation to be analysed and benchmarked DoubleCrunch performance. On the Tesla K40m card the simulation performed at 1.615 GigaNumbers per second. Both Cephes and the accurate CRLibM implementations of exp are essentially branch-free, thus we assume that this performance level would be sustained on the whole domain of exp function, which includes $\approx 1.008 \cdot 2^{63}$ points. From this data, and assuming perfect scaling proportionally to the number of FLOPS, we estimate the compute time on different systems. Results are presented in Tab. 4.

[3]On AMD GCN architecture addition is cheaper than FMA, but to preserve bit-exactness we use the same scheme as on other accelerators

[4]It also contains expm1 accurate to 120 bits.

**Table 4: Compute time for brute-force error analysis of exp function on full range**

| Compute system | Compute time |
|---|---|
| Tesla K40m | 183 years |
| 16× Tesla K80 | 5 years |
| 1 PFLOPS GPU cluster | 3 months |
| Titan cluster | 3.5 days |
| 1 EFLOPS GPU cluster | 2 hours 17 minutes |

## 7. CONCLUSION

We explored the possibility of exhaustive search for worst-case errors of double-precision floating-point expressions and described a software framework to aid such brute-force analysis. Despite the enormously huge search space, some practical problems can be solved even on a single machine with consumer-grade GPU.

## Acknowledgment

We thank Edmond Chow and his Intel Parallel Computing Center at Georgia Tech for access to Xeon Phi-based platforms, as well as the NVIDIA CUDA Center of Excellence for access to NVIDIA hardware. This material is based upon work supported by the U.S. National Science Foundation (NSF) Award Number 1339745, and CAREER Award Number 0953100. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, Intel, or NVIDIA.

## 8. REFERENCES

[1] Sollya: An environment for the development of numerical codes, author=Chevillard, Sylvain and Joldeş, Mioara and Lauter, Christoph, booktitle=Mathematical Software–ICMS 2010, pages=28–31, year=2010, publisher=Springer.

[2] D. H. Bailey and J. M. Borwein. *Experimental applied mathematics*. 2012.

[3] S. Chevillard and C. Lauter. A certified infinite norm for the implementation of elementary functions. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pages 153–160. IEEE, 2007.

[4] N. Corporation. *NVIDIA CUDA C Programming Guide*, 2014.

[5] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. Cr-libm a library of correctly rounded elementary functions in double-precision, 2009.

[6] F. De Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *Computers, IEEE Transactions on*, 60(2):242–253, 2011.

[7] P. Fortin, M. Gouicem, and S. Graillat. GPU-accelerated generation of correctly-rounded elementary functions. *arXiv preprint arXiv:1211.3056*, 2012.

[8] S. Graillat, P. Langlois, and N. Louvet. Compensated horner scheme. 2005.

[9] S. Graillat, P. Langlois, and N. Louvet. Improving the compensated horner scheme with a fused multiply and add. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1323–1327. ACM, 2006.

[10] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 111–118. IEEE, 2001.

[11] P. Markstein. *IA-64 and elementary functions: speed and precision*. Prentice Hall, 2000.

[12] S. L. B. Moshier. *Methods and programs for mathematical functions*. Halsted Press, 1989.

[13] E. S. Raymond. *The art of UNIX programming*. Prentice-Hall, 2003.