Poseidon: Profile-Guided Numerical Rewriting at Full-Application Scale

Siyuan Brant Qian
University of Illinois Urbana-Champaign
Champaign, USA
siyuanq4@illinois.edu

Vimarsh Sathia
University of Illinois Urbana-Champaign
Champaign, USA
vsathia2@illinois.edu

Ivan R. Ivanov
Institute of Science Tokyo
RIKEN CCS
Kobe, Japan
ivanov.i.e641@m.isct.ac.jp

Jan Hückelheim
Argonne National Laboratory
Lemont, USA
jhueckelheim@anl.gov

Paul Hovland
Argonne National Laboratory
Lemont, USA
hovland@anl.gov

William S. Moses
University of Illinois Urbana-Champaign
Champaign, USA
wsmoses@illinois.edu

Abstract

Floating-point numbers are finite-precision approximations to real numbers and are ubiquitous in computer applications in nearly every field. Selecting the right floating-point representation that balances performance and numerical accuracy is a difficult task, one that has become even more critical as hardware trends toward high-performance, low-precision operations. Although a plethora of numerical techniques, including algebraic rewriting and mixed-precision tuning, can help navigate the tradeoff between performance and accuracy, applying complex numerical optimizations to real-world code is still a challenging engineering task that requires mathematical expertise and application-specific numerical context. We present Poseidon, a modular and extensible framework that fully automates floating-point optimizations for real-world applications within a production compiler. Within a production compiler, Poseidon auto-extracts symbolic expressions from pre-optimized IR, derives their numerical context from a small surrogate profile, and composes candidates from diverse techniques (e.g., Herbie's e-graph-based algebraic rewrites and ADAPT-style precision tuning) via a shared dynamic-programming solver. On a quaternion differentiator, Poseidon composes algebraic rewriting with precision tuning for a 1.46× speedup at relative error of 10^{-7}. On DOE's proxy application LULESH, Poseidon recovers optimal FP64 accuracy without substantially reducing performance.

1 Introduction

Real-number computations are critical in nearly every scientific discipline, ranging from biology, physics, and chemistry to machine learning and beyond. Representing and performing real-number computations in their exact form is often impossible or prohibitively expensive. In practice, many users of real arithmetic approximate their programs using floating-point (FP) values, which represent numbers in scientific notation with a fixed-length exponent and mantissa. Consequently, the use of FP numbers in storage and computation can lead to results which differ from their exact theoretical values in real arithmetic. Practitioners who use FP numbers are often unaware of its consequences: subtle accuracy bugs have surfaced in libraries like TensorFlow Probability, PyMC, and Stan [1, 3, 7]. Tools like Adapt [12], FPTuner [4], Precimonious [15], HiFPTuner [9], and GPUMixer [11] can analyze large parts of applications, but are limited to just changes in precision. Tools like Herbie [13], Daisy [6], Salsa [5], and MegaLibm [2] perform more advanced rewrites of FP expressions, but are limited to individual DSL expressions or math library functions and provide limited support for general-purpose language constructs. Optimizing real-world numerical code is difficult along three axes: *context*, *scope*, and *scale*.

Context. The legality and benefit of a numerical rewrite depend on the numerical context the program operates in. Fig. 1 shows a common accumulation loop: for each element of a 10,000-element array, the program applies an exponential and a sigmoid, then adds the result to a running sum. Lowering only the inner exp calls to FP32 yields a 1.6× speedup with relative error around 10^{-12}; lowering the accumulator instead causes error to increase to 10^{-4} due to rounding error. Profiling just 10 iterations reveals that the accumulator is $10^4\times$ more sensitive than the activations under an ADAPT-style metric [12]. We propose the *structured critical value hypothesis*: which values are accuracy-critical is often structural (e.g., reductions, cancellations, thresholds, one-hot vectors). We argue that a small surrogate profile often suffices to guide numerical rewrites.

Scope. Multiple numerical rewriting techniques, including precision tuning and algebraic rewriting, help navigate the performance/accuracy tradeoff. Consider $(10^8 + 1) - 10^8$. In FP32, the 1 is rounded away in the addition, yielding 0. FP64 preserves the result but pays an order of magnitude

```
double accumulate(double *data, size_t n) {
  double sum = 0.0;
  // float sum = 0.0f;                      Error += 10^-4
  for (size_t i = 0; i < n; ++i) {
    double val = sigmoid(exp(data[i]));
    // float val = sigmoidf(expf(data[i])); Error += 10^-12
    sum += val;
  }
  return sum;
}
```

Figure 1. A common accumulation pattern. Lowering only `exp` to FP32 keeps error near 10^{-12}; lowering the accumulator inflates the error to 10^{-4}.

in cost. Algebraic rewrites circumvent the tradeoff in this example – reassociating the same expression in FP32 to $(10^8 - 10^8) + 1$ yields 1 at FP32 cost. Applying them to a program requires numerical context (e.g., relative magnitudes of the operands) and expertise in numerical analysis. Tools like Herbie [13], which use equality saturation over an e-graph to discover such rewrites, partially automate this, but only on isolated DSL expressions. Furthermore, composing algebraic rewriting with precision tuning produces Pareto-optimal programs that neither technique can enable in isolation, as we demonstrate on a quaternion differentiator in Sec. 3.

Scale. In real applications, the mathematical expressions of interest are often scattered across control flow, memory traffic, and function boundaries. As Fig. 2 illustrates, a production compiler provides the necessary machinery to optimize a real-world numerical application. Operating within the compiler grants two key advantages. First, a two-phase PGO-like pipeline can automate end-to-end FP optimization of large-scale applications without user intervention to provide numerical context, reason about the legality, or evaluate the performance/accuracy implications of rewrites. Second, acting within the compiler enables interoperability with compiler analyses and optimizations: pre-optimizations like `mem2reg`, `SimplifyCFG`, `InstCombine`, inlining, and loop unrolling remove control flow and data movement through memory, exposing a more complete view of the underlying numerical computation than is available to source-level or binary-level tools.

2 The Poseidon Framework

We present Poseidon, a framework that addresses these three challenges by performing automated numerical rewrites within a production compiler. Fig. 3 illustrates Poseidon's workflow. Poseidon operates as a two-phase compiler. The first compilation runs a profiling pass that instruments each FP instruction to collect numerical context such as execution counts, value ranges, and gradients; we also track an ADAPT-style sensitivity metric [12] to quantify precision-change impact. The second compilation uses pre-optimizations (e.g., `mem2reg`, `SimplifyCFG`, inlining, loop unrolling) to simplify memory traffic and control flow, and extracts *FP subgraphs* by scanning the def-use graph of the resulting intermediate representation. The candidate generation phase consumes collected numerical context and proposes algebraic rewrites (via Herbie [13]) and precision changes (ranked by ADAPT sensitivity). A subsequent candidate evaluation phase scores each candidate's performance/accuracy impacts offline using an internal cost model and an MPFR [8]-based accuracy model. A DP solver builds a global cost-error frontier by composing the generated numerical rewrites. Post-optimizations (CSE, DCE, vectorization) further improve the IR. We refer to our full paper [14] for details.

Ongoing: GPU extension. AI-optimized datacenter GPUs are deprioritizing hardware FP64 throughput. Existing scientific kernels, however, are often written in FP64, raising concerns that directly downcasting to FP32 substantially hurts numerical accuracy. Poseidon's techniques generalize to GPU kernels. The framework can integrate advanced numerical algorithms such as branch-free extended-precision arithmetic [16] to achieve near-FP64 accuracy using only FP32 operations; e-graph-based algebraic rewrites can further repair accuracy loss introduced by precision changes.

3 Evaluation

We evaluate Poseidon on multiple case studies to evaluate its effectiveness. We run Poseidon's profiling pass on small surrogate input sets (10–1000 points) and evaluate optimized programs on disjoint test sets of 100k–1M inputs. We quantify accuracy using geomean and worst-case relative errors against MPFR [8] references with at least 512 mantissa bits.

3.1 Composition Pays: Quaternion Differentiator

We optimize `dquat`, a program that computes the derivative of quaternions with respect to input exponential maps. `dquat` contains multiple loops iterating through vector elements via memory accesses to compute norms and scalar products. Poseidon leverages `mem2reg`, SROA, and loop unrolling to extract l^2-norm expressions like $\theta = \sqrt{x^2 + y^2 + z^2}$ and Jacobian-entry expressions like $u^2(0.5\cos(\theta/2) - \sin(\theta/2)/\theta) + \sin(\theta/2)/\theta$.

Poseidon's DP solver effectively composes precision changes (PT) with algebraic rewrites (AR) to find tradeoffs that neither finds in isolation (Fig. 4). For example, composing both leads to a 1.46× speedup at geomean relative error 6.23×10^{-7} by lowering nonlinear operations to FP32 and replacing $u^2(0.5\cos(\theta/2) - \sin(\theta/2)/\theta) + \sin(\theta/2)/\theta$ with $\sin(\theta/2)/\theta$; this rewrite is legal only because the profile reveals $0 < \theta < 0.2$ and $u^2 < 1$. AR alone also lets Poseidon select 3 Herbie rewrites that add special handling for small vector components, reducing geomean relative error by 26.8% at 1.13× compute.

```c
// Source code
double foo(double x) {
  if (x > 0)
    return pow(x, 3);
  else
    return 0.0;
}
```

```
; -O3
entry:
  %cmp = fcmp ogt double %x, 0.0
  br i1 %cmp, label %if.then, label %return
if.then:
  %call = call double @pow(double %x, double
  ↪ 3.0)
  br label %return
return:
  %r = phi double [ %call, %if.then ], [ 0.0,
  ↪ %entry ]
  ret double %r
```

```
; -O3 -ffast-math + Pre-Process
entry:
  %p = call fast double @llvm.powi.f64.i32(
         double %x, i32 3)
  %r = call fast double @llvm.maxnum.f64(
         double 0.0, double %p)
  ret double %r
```

Figure 2. Standard compiler analyses and optimizations (`SimplifyCFG`, `mem2reg`, inlining) eliminate branches and expose the underlying math. Poseidon replaces `foo` with a single `max(0,pow(x,3))` expression that downstream callers can fold further.

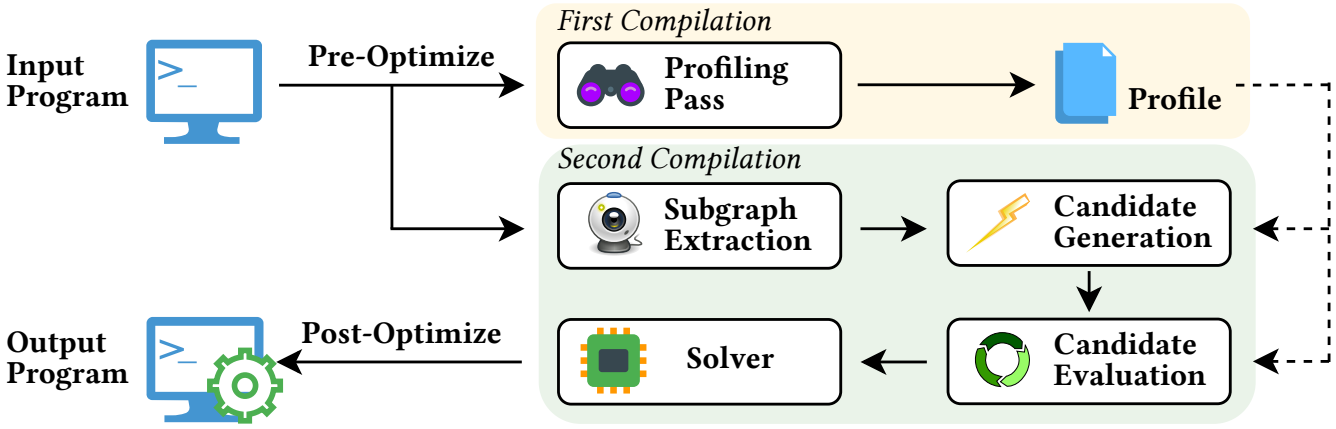

Figure 3. An overview of Poseidon's workflow for automated FP rewrites within the compiler. Poseidon first instruments the pre-optimized program to generate FP profiles. Then it synthesizes and evaluates candidate rewrites, and solves for optimal performance/accuracy tradeoffs. After materializing rewrites, post-optimizations further improve the IR.

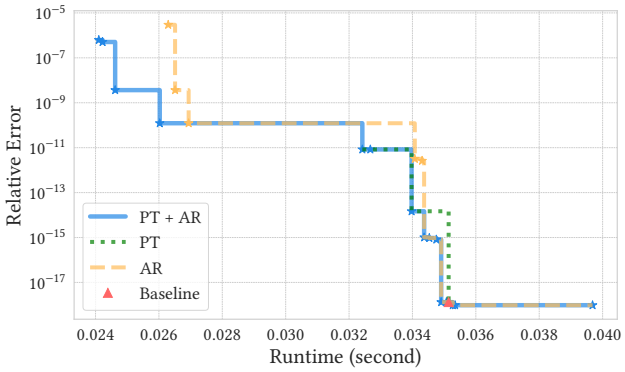

Figure 4. Pareto fronts for `dquat`. Solid blue = Precision Tuning (PT) + Algebraic Rewriting (AR), dashed yellow = AR only, dotted green = PT only. Red triangle is the baseline; stars are optimized programs. *Lower-left is better.*

3.2 Scale: LULESH

We evaluate Poseidon on LULESH [10], a DOE proxy application that simulates Lagrangian hydrodynamics. LULESH contains over 5,600 lines of code and over 200 `for` loops. The original FP64 implementation gradually loses precision: for problem size 50, the computed final origin energy deviates from a 256-bit MPFR reference by 12 units in the last place (ULPs) after 1662 iterations. We generate a small surrogate profile from 100 iterations on problem size 30 (≈1.3% of tested computation) to guide Poseidon's optimization pass.

As shown in Fig. 5, Poseidon finds a performance/accuracy tradeoff that matches MPFR reference results up to the representational limits of FP64 (i.e., 0 ULP error) without substantial runtime increase, using 9 accuracy-improving and 2 simplifying algebraic rewrites produced by Herbie [13]. This optimized FP64 program is more accurate *and* faster than a hardware FP80 variant, which is 6.79× slower than the FP64 baseline yet still 2 ULPs off. While MPFR floats can produce more accurate results, an MPFR implementation is at least 500× slower.

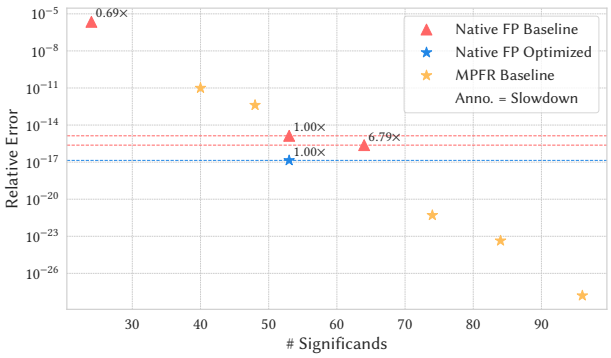

Figure 5. Relative errors vs. number of significands used for size-50 LULESH runs (*lower is better*). The blue star marks the optimized FP64 program, yellow stars mark MPFR baselines, and red triangles mark native baselines with `float`, `double`, and `x86_fp80`, from left to right. Annotations show runtime ratios relative to the `double` baseline (*lower is better*).

4 Conclusion

We have presented Poseidon, an end-to-end framework that addresses the challenges of automated numerical rewrites of full-scale scientific applications. Poseidon eliminates the need for human expertise of numerical methods and understanding of specific numerical context required to perform correct and performant rewrites. On a quaternion differentiator and LULESH, Poseidon delivers outsized performance/accuracy benefits from small surrogate profiles.

References

[1] Oriol Abril-Pla, Virgile Andreani, Colin Carroll, Larry Dong, Christopher J Fonnesbeck, Maxim Kochurov, Ravin Kumar, Junpeng Lao, Christian C Luhmann, Osvaldo A Martin, et al. 2023. PyMC: a modern, and comprehensive probabilistic programming framework in Python. *PeerJ Computer Science* 9 (2023), e1516.

[2] Ian Briggs, Yash Lad, and Pavel Panchekha. 2024. Implementation and Synthesis of Math Library Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 32 (Jan. 2024), 28 pages. doi:10.1145/3632874

[3] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76 (2017).

[4] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 300–315. doi:10.1145/3009837.3009846

[5] Nasrine Damouche and Matthieu Martel. 2018. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs. In *Automated Formal Methods (Kalpa Publications in Computing, Vol. 5)*, Natarajan Shankar and Bruno Dutertre (Eds.). EasyChair, 63–76. doi:10.29007/j2fd

[6] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems* (Porto, Portugal) *(ICCPS '18)*. IEEE Press, 208–219. doi:10.1109/ICCPS.2018.00028

[7] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv preprint arXiv:1711.10604* (2017).

[8] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007).

[9] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 333–343. doi:10.1145/3213846.3213862

[10] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[11] Ignacio Laguna, Paul C. Wood, Ranvijay Singh, and Saurabh Bagchi. 2019. GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications. In *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11501)*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer, 227–246. doi:10.1007/978-3-030-20656-7_12

[12] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 614–626. doi:10.1109/SC.2018.00051

[13] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/2737924.2737959

[14] Siyuan Brant Qian, Vimarsh Sathia, Ivan R. Ivanov, Jan Hückelheim, Paul Hovland, and William S. Moses. 2026. Thinking Fast and Correct: Automated Rewriting of Numerical Code through Compiler Augmentation. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 548–562. doi:10.1109/CGO68049.2026.11395228

[15] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 27, 12 pages. doi:10.1145/2503210.2503296

[16] David Kai Zhang and Alex Aiken. 2025. High-Performance Branch-Free Algorithms for Extended-Precision Floating-Point Arithmetic. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25)*. Association for Computing Machinery, New York, NY, USA, 695–710. doi:10.1145/3712285.3759876