Thinking Fast and Correct: Automated Rewriting of Numerical Code through Compiler Augmentation

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 the common wisdom around changing floating-point precision implies that accuracy and performance are inversely correlated, more advanced techniques can often circumvent this tradeoff. Applying complex numerical optimizations to real-world code, however, is an arduous engineering task that requires expertise in numerical analysis and performance engineering, and application-specific numerical context. While there is a plethora of existing tools that partially automate this process, they are limited in the scope of optimization techniques or still require substantial human intervention. We present Poseidon, a modular and extensible framework that fully automates floating-point optimizations for real-world applications within a production compiler. Our key insight is that a small surrogate profile often reveals sufficient numerical context to drive effective rewrites. Poseidon operates as a two-phase compiler: the first compilation instruments the program to capture numerical context; the second compilation consumes profiled data, generates and evaluates candidate rewrites, and solves for optimal performance/accuracy tradeoffs. Poseidon's interoperability with standard compiler analyses and optimizations grants it analysis and optimization advantages unavailable to existing source- and binary-level approaches. On multiple large-scale applications, Poseidon leads to outsized benefits in performance without substantially changing accuracy, and outsized accuracy benefits without diminishing performance. On a quaternion differentiator, Poseidon enables a $1.46\times$ speedup with a relative error of 10^{-7}. On DOE's LULESH hydrodynamics application, Poseidon improves program accuracy to exactly match a 512-bit simulation run without substantially reducing performance.**

***Index Terms*—Floating-point arithmetic, Numerical analysis, Optimizing compilers, Performance analysis, Scientific computing**

I. Introduction

Real-number computations are critical in nearly every scientific discipline, ranging from biology [1, 2], physics [3, 4], and chemistry [5, 6] to machine learning [7, 8] and beyond. Representing and performing real-number computations in their exact form is often impossible or prohibitively expensive. For example, $\sqrt{2}$ is an irrational number and cannot be exactly represented with a finite decimal or binary expression. 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. Efficiently and accurately computing real-valued functions is a well-studied domain and has led to the birth of the numerical analysis field.

Practitioners who use FP numbers are often unaware of the consequences of FP approximations in their programs, and accidentally end up with numerically incorrect results. In fact, even libraries that pay careful attention to numeric results like TensorFlow Probability [9], PyMC [10], and Stan [11] have all had bugs generating incorrect or biased results due to incorrect FP approximations over the years[1]. In contrast, many domains like machine learning can tolerate some degree of inaccuracy. For instance, the fact that gradient descent is effective even with approximate gradients has motivated researchers to find tradeoffs in FP computations, e.g., intentionally reducing numerical precision to accelerate training. Recent years have witnessed shifts from machine learning models using 32-bit floats to those using 16-bit representations [12], 8-bit representations [13], 4-bit representations [14], and recently even single-bit (really 1.58 bits) models [15].

Simply reducing precision until the loss stops decreasing is an often-used and usually sufficient approach to maintaining accuracy. However, this method often sacrifices a substantial amount of performance improvements. For example, consider the computation in Fig. 1 which computes $10^8 + 1 - 10^8$.

[1] See https://github.com/tensorflow/probability/issues/542, https://github.com/pymc-devs/pymc/issues/1787, and https://github.com/stan-dev/stan/issues/2178 for examples of correctness errors due to FP approximations.

979-8-3315-9288-2

Computation: $(10^8 + 1) - 10^8 = ?$

FP32:
```
(1.0000000e+08 + 1.0000000e+00) - 1.0000000e+08
1.0000000e+08 - 1.0000000e+08
0.0000000e+08
```

FP64:
```
(1.000000000000000e+08 + 1.000000000000000e+00) - 1.000000000000000e+08
1.000000010000000e+08 - 1.000000000000000e+08
1.000000000000000e+00
```

FP32 (Reassoc.):
```
(1.0000000e+08 - 1.0000000e+08) + 1.0000000e+00
0.0000000e+00 + 1.0000000e+00
1.0000000e+00
```

Fig. 1: An example of how different FP precisions and orders of operations affect the evaluation of $10^8 + 1 - 10^8$. In FP32 arithmetic, the 1 term is lost when added to 10^8 due to limited precision, resulting in an incorrect result of 0. However, FP64 arithmetic preserves the 1 term due to its extended precision, which leads to the correct result. Reassociating the operations in FP32 can also mitigate this numerical inaccuracy, but requires numerical context such as relative magnitudes.

```
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;
}
```

Fig. 2: A common accumulation pattern in training loops (see [26] for details) and scientific solvers where each iteration computes a nonlinear term (`val`) and reduces it into an accumulator (`sum`). On a 10,000-element array, the FP64 baseline yields a relative error of around 10^{-15}. Lowering only exp operations (one within `sigmoid`) to FP32 delivers a $1.6\times$ speedup while keeping the final error near 10^{-12}, whereas lowering the addition to `sum` results in a much larger error of 10^{-4} due to rounding.

With 32-bit FP arithmetic (FP32), the small magnitude of 1 relative to 10^8 makes $10^8 + 1$ numerically indistinguishable to 10^8, leading to an incorrect result. Adding additional bits of precision, in this case the use of a 64-bit float (FP64), allows $10^8 + 1$ to be represented exactly, thus correcting the overall computation. Alternatively, by simply re-associating the terms to perform $(10^8 - 10^8) + 1$ instead, in other words, grouping the large numbers together, enables the computation to succeed without the need to increase precision.

More advanced transformations, such as algebraic rewrites, can bring significant benefits to both the runtime performance and accuracy of programs. However, applying these techniques to real-world codes is significantly more challenging as their successful application requires context about FP expressions. For example, applying the reassociation in Fig. 1 requires knowing the relative magnitude of the operands (i.e., that we were computing a big number minus a small number plus a big number). Alternatively, one would go the other direction and simplify an expression by removing a term if its individual contribution to the final result is negligible. Consider the expression $f(x) = 0.000001 \cdot \text{abs}(\sin(x)) + \exp(\text{abs}(x))$. Since the final result is always at least one, omitting the first term will make at most a 0.0001% difference.

Effectively tuning the performance/accuracy tradeoff therefore requires simultaneous knowledge of: (1) rewrite techniques from numerical analysis, (2) numerical context on the ranges and sensitivities of values, and (3) the computational cost and accuracy of potential expressions. Expecting programmers to have expertise in numerical analysis alone is quite a tall order, let alone all of the above. The difficulty of performing rewrites by hand begs the key question:

1) How much performance and accuracy is sacrificed due to suboptimal choices of FP precision and expressions?

Several tools have been developed to partially automate this re-tuning, but they are either limited in the scope of transformations or require substantial human intervention. Tools like Adapt [16], FPTuner [17], Precimonious [18], HiFPTuner [19], and GPUMixer [20] can analyze large parts of applications, but are limited to just changes in precision. Tools like Herbie [21], Daisy [22], Salsa [23], and MegaLibm [24] perform more advanced rewrites of FP expressions, but are limited to individual DSL expressions or math library functions and provide limited support for control-flow constructs. STOKE [25] performs binary-level stochastic FP optimization, but it requires substantial hyperparameter tuning to generate useful program and has limited support for control flow, memory dereferences, and math library functions. Automatically applying numerically context-sensitive FP rewrites to large-scale programs is difficult for several reasons. First, algebraic rewriting tools require precise understanding of FP computations in the form of symbolic expressions. These expressions are often scattered across programs, passing through complex execution context like control flow, data structures, and memory accesses. Extracting the underlying symbolic expressions from large-scale programs requires manual inspection of source code. Second, numerical context such as values and sensitivities is required to generate effective candidate expressions (consider Fig. 1) – again requiring human intervention. Given that existing FP optimization techniques either require substantial human intervention or are limited in the scope of transformations, it begs a second key question:

2) Can advanced context-sensitive FP optimization techniques be automated to reduce the need for human intervention while remaining effective?

Our key insight to answer this question is that a small surrogate profile often reveals enough numerical context for context-sensitive FP rewrites. For example, profiling just 10 iterations in Fig. 2 allows an automated tool to conclude that the addition is $10 - 10^4\times$ more sensitive to precision changes (using ADAPT metrics [27]) than other operations; an automatic tool should therefore treat the addition as accuracy-critical while safely lowering the expensive exp operations to FP32. This rewrite leads to a $1.6\times$ speedup without substantial accuracy loss, whereas naively reducing the precision of entire program (and thus the addition), leads to inaccurate results. While the examples in Fig. 1 and 2 clearly demonstrate the need for profiling information to ensure effective optimization, perfect information about a realistic application's inputs and runtime values is often infeasible. In this work, we hypothesize that a small surrogate profile is sufficient to gain most of the performance/accuracy benefits. To justify this, we propose the ***structured critical value hypothesis*** – whether a value is critical to the overall accuracy of a floating-point computation is often structural and does not change with real-world test cases. For example, consider `sum` from Fig. 2. On most real-world datasets, the input data would cause the value being summed to be reasonably distributed and maintaining the accuracy of `sum` is critical. However, there may be some classes of inputs, for example if data is the result of a one-hot vector – where the `sum` is not critical and the accuracy of `val` expression itself is key. As long as we have approximate profiling information to determine if the input data is well-distributed or one-hot and decide which values are critical, the exact ranges of every value in the profile do not matter.

This insight motivates us to answer both questions with **Poseidon**, a framework for automated context-sensitive FP optimizations. Poseidon operates as a two-phase compiler. The first compilation instruments a user program to generate FP profiles with instruction values and sensitivities (Sec. III-B). The second compilation automatically analyzes FP computation (Sec. III-C), synthesizes (Sec. III-D) and evaluates (Sec. III-E) FP rewrites, and solves (Sec. III-F) for useful performance/accuracy tradeoffs within the program. To answer our first scientific question, we incorporate expression rewrites from Herbie [21] and changes of FP precision into the second compilation. Poseidon is designed to be modular and fully extensible, enabling arbitrary FP rewrites, solvers, and cost models to be added with ease.

To our knowledge, we are the first to perform context-sensitive FP rewrites *within a production compiler.* Operating within the compiler grants Poseidon two key advantages. First, Poseidon's two-phase PGO-like pipeline automates 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 lets Poseidon interoperate with compiler analyses and optimizations. Specifically, Poseidon is able to leverage pre-optimizations, e.g. `mem2reg`, `SimplifyCFG`, `InstCombine`, inlining, and loop unrolling, that remove control flow and data movement through memory, enabling it to extract larger FP subgraphs than would be available in source codes (which lacks such optimizations) or binaries (after which the infinite SSA virtual register state is lowered into more memory loads and stores). In Fig. 3, standard compiler optimizations allow Poseidon to eliminate branches wherever possible and replace the entire function with a single `max(0,pow(x,3))` expression, which can be inlined and further folded into surrounding expressions. To handle control flow, existing tools like Salsa [23] must consider every branch as a separate case in their analyses, resulting in limited numerical context and prevents cross-branch optimization. Poseidon also benefits from downstream optimizations such as GVN, CSE, DCE, and vectorization, to amplify its gain from FP rewrites. To our knowledge, no other framework can directly impact code generation through pre- and post-optimizations as Poseidon does.

Overall, we present the following contributions:

- The first end-to-end framework for automated context-sensitive FP rewrites within a production compiler.
- A novel dynamic-programming algorithm that solves for optimal performance/accuracy tradeoffs within a large-scale application.
- Evaluations showing that Poseidon performs effective FP optimizations given small surrogate profiles. *To our knowledge, we are the first to automate context-sensitive FP rewrites on a full-application scale.*
- Ablation studies on the design of Poseidon's components.
- A discussion on the importance of interoperability with compiler analyses and optimizations in FP optimization.

II. Background

A. Floating-Point Numbers

Let us consider IEEE 754 FP numbers in double and single precision [28]. Single-precision numbers use 32 bits to represent a number with 1 sign bit, 8 exponent bits, and 24 mantissa bits, of which only 23 need to be explicitly stored. When represented as a decimal number, this leads to approximately 7 meaningful decimal places and a maximum number of approximately 10^{38}.

Double-precision numbers use 64 bits, allowing for 11 exponent bits and 53 mantissa bits, of which 52 are explicitly stored, leading to approximately 15 significant decimal places of precision and a maximum number of approximately 10^{308}.

As shown in Fig. 1, using a more precise FP format often reduces roundoff errors, as the intervals between FP numbers are smaller. On the other hand, more precise formats use more memory, and often require more time and energy for each operation, presenting a tradeoff between precision and performance. We refer to [29] for an overview of IEEE FP formats, and to [30] for a survey about the tradeoff between accuracy and speed in various applications.

B. Floating-Point Error Estimation

Manipulating the performance/accuracy tradeoff requires estimates or ideally bounds on the error incurred at a given precision. A straightforward approach would be to apply interval arithmetic to determine the possible ranges of input values for each scalar operation, and to compute worst-case roundoff error estimates for that operation for the given input ranges. As a refinement, interval arithmetic can be replaced with more precise affine arithmetic [31, 32] or model checking [33].

Rigorous bounds, while excellent for simple codes, are often too large to be useful in real-world applications. For example, the roundoff error incurred when evaluating the simple expression $\frac{1}{x-y}$ can be unbounded, especially if the ranges of x and y

```
// 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
```

Fig. 3: Interoperability with standard compiler analyses and optimizations enable Poseidon to eliminate control flow wherever possible. With branches gone, Poseidon replaces the entire function `foo` with a single `max(0,pow(x,3))` expression which can be inlined and further folded into surrounding expressions.

overlap. Dynamic approaches that provide roundoff error estimates for a given input, set of inputs, or input ranges can often provide less pessimistic and more useful, albeit not guaranteed, error estimates. Some popular approaches combine local error metrics for individual operations with a linearized model of the overall computation to compute the cumulative effect of errors on the final result, or to compute the sensitivity of the final result with respect to the precision of individual operations or operands [27, 34, 35]. Other approaches rely on probabilistic roundoff error estimators [36, 37].

C. Expression Rewriting

FP arithmetic violates associativity and distributivity, which means that certain rewrite rules that transform an expression into an equivalent expression under real arithmetic, may increase or reduce roundoff errors under FP arithmetic. A classic example is the summation of a vector of numbers. While the order of summation is irrelevant under real arithmetic, multiple strategies including pairwise summation or ordered summation have been studied in the literature and have different performance characteristics and roundoff error bounds [38]. Tools like Herbie [21] have been developed to automatically discover expression rewrites that improve accuracy, and the approach has been extended to support a limited form of control flow within a domain-specific language (DSL) [23].

It is also possible to replace an expression with a different one that would not be equivalent in real arithmetic, but leads to similar or even bitwise-exact results in FP arithmetic. For example, the expression $\sin(x)$ can be replaced with x for sufficiently small values of x, or higher-order Taylor polynomials for larger values of x. Range reduction techniques can be used especially for periodic functions to project inputs into a range in which such approximations are most accurate, or the best approximation can be chosen depending on the input. Commonly used implementations of math libraries internally use these approaches to compute, e.g., transcendental functions [39], and cheap approximations for some important functions have been developed in past work [40].

Existing work has found that expression rewriting can sometimes be combined with FP precision tuning to achieve a better performance/accuracy tradeoff [22].

D. Floating-Point Precision Tuning

While it is sometimes enough to globally choose either single or double precision, a better performance/accuracy tradeoff is often achievable by selecting the precision of each variable or operation individually. Mixed-precision programs are more challenging to analyze, and programs that attempt to identify the best mixed-precision configuration for given performance/accuracy constraints and objective functions have to explore an exponentially large search space. Previous works have therefore discussed a number of tools; we refer to [41] for a survey, including genetic algorithms [42], hierarchical searches [43], stochastic tuning [25], or by using the structure of individual applications [44].

III. Design

We present Poseidon, a framework for end-to-end context-sensitive FP optimizations within a production compiler. Fig. 4 provides an overview of the framework. Poseidon first runs a profiling pass to compile an instrumented program and executes it to generate FP profiles (Sec. III-B). Given the profiles, Poseidon's optimization pass identifies FP subgraphs within the input program (Sec. III-C), generates candidate FP rewrites (Sec. III-D), and evaluates their performance/accuracy impacts on the overall program (Sec. III-E). Finally, Poseidon uses a dynamic-programming (DP) solver to compute the performance/accuracy frontier within the program (Sec. III-F).

A. Pre-Optimizations

Poseidon's profiling and optimization passes operate on the same pre-optimized IR. Poseidon runs standard compiler analyses and optimizations (see Fig. 3 and Sec. IV) to eliminate control flow and promote memory accesses into SSA form, exposing larger FP subgraphs and richer numerical context needed to enable performant and accurate context-sensitive FP rewrites (see Sec. III-C). Pre-optimizations grant Poseidon the key advantage of capturing numerical context unavailable to source- and binary-level approaches.

B. Profiling Pass

The first compilation of Poseidon collects numerical context with a profiling pass to augment input programs with gradient computation (synthesized by Enzyme's reverse-mode AD) and profiler calls that record instruction values and gradients. Fig. 5

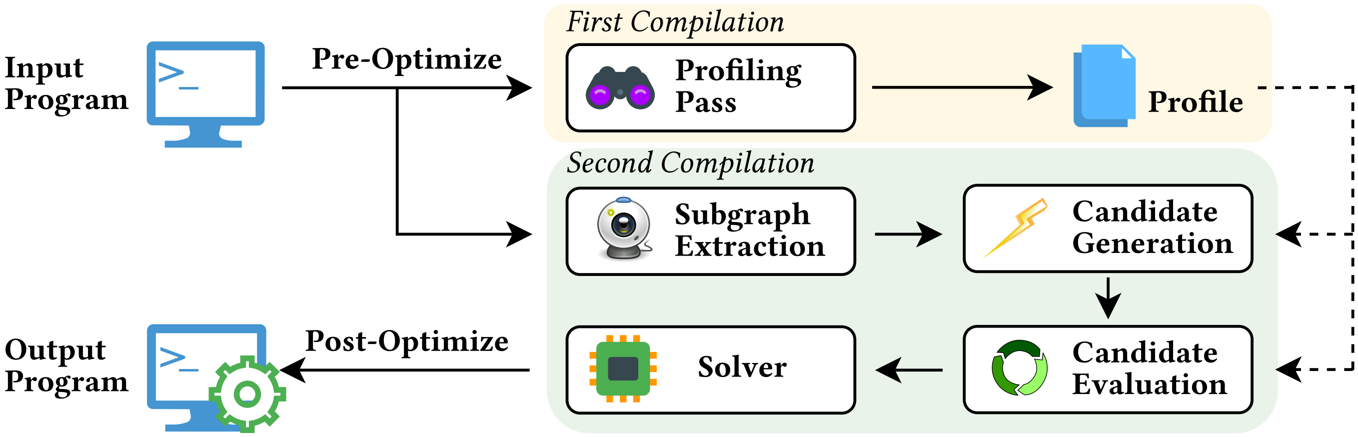

Fig. 4: An overview of Poseidon's workflow for automated FP rewrites within the compiler. Poseidon first instruments the pre-optimized program to generate FP profiles. Then it synthesizes and evaluates candidate rewrites, and solves for optimal performance/accuracy tradeoffs. After materializing rewrites, post-optimizations further improve the IR.

```
define double @instrumented_f(double %x, double %seed) {
entry:
  %0 = alloca [1 x double], align 8
  %1 = call fast double @llvm.sin.f64(double %x) ; idx = 0
  store double %x, ptr %0, align 8
  call void @enzymeLogValue(i64 0, double %1, i32 1, ptr %0)
  call void @enzymeLogGrad(i64 0, double %1, double %seed)
  %2 = call fast double @llvm.cos.f64(double %x)
  %3 = fmul fast double %seed, %2
  ret double %3
}
```

Fig. 5: Instrumented LLVM IR for $f(x) = \sin(x)$. Highlighted lines are inserted by Poseidon's profiling pass: they compute gradients and pass runtime information to the profiler.

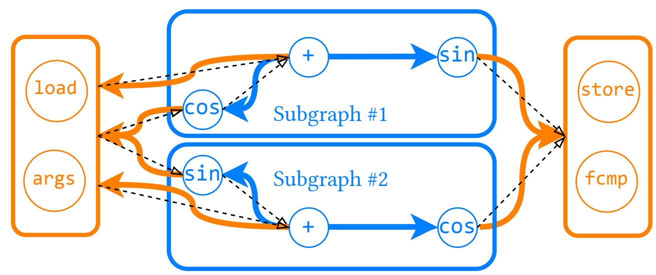

Fig. 6: Poseidon's flood-fill algorithm for subgraph extraction. Dotted black arrows indicate data dependences.

shows an example of instrumenting a function that evaluates $f(x) = \sin(x)$. Within the profiler, Poseidon maintains instruction execution counts, operand bounds, and running sums of observed values and gradients. Gradients measure how much numerical error in intermediate value v propagates to the final result y: under first-order approximation, a local perturbation Δv (e.g., induced by rounding, precision changes, or alternative algebraic forms) yields an output change $\Delta y \approx \frac{\partial y}{\partial v}\Delta v$. We additionally track an ADAPT-style metric [27] by accumulating $|\frac{\partial y}{\partial v} v|$, which measures the impact of precision changes. Beyond pre-optimizations that simplify control flow, profiling also captures control-flow behavior implicitly: execution counts and gradients accumulate only along executed paths, so frequently executed computations naturally receive larger aggregated sensitivity. For example, if a `sum += x * x;` statement executes 1000 times, each iteration contributes `grad_sum` to `fmul`'s gradient and `2.0 * x * grad_sum` to `x`'s gradient, resulting in a higher estimated cost/accuracy impact than less frequently executed statements (see Sec. III-E).

C. Subgraph Extraction

The first step of the optimization pass extracts *FP subgraphs*, i.e., connected components of FP instructions, from pre-optimized IR. Poseidon supports most FP arithmetic operations, including basic operations (`+`, `-`, `*`, `/`) and all LIBM math library functions. As we illustrate in Fig. 6, Poseidon performs a worklist-based flood-fill traversal of the pre-optimized IR: a worklist is initialized with a seed FP instruction, and each iteration dequeues an FP arithmetic instruction, marks it as part of the current FP subgraph, and finally enqueues its operands and users. This traversal terminates when the worklist becomes empty. This linear-time traversal simultaneously 1) identifies FP subgraphs, which enables one-pass construction of FP expressions, and 2) finds *input values* that feed each FP subgraph and *output instructions* that propagate FP results out of subgraphs. Although Poseidon's subgraph extraction phase does not include `br`, `load`, and `store` instructions as part of subgraphs, pre-optimizations (Sec. III-A) greatly simplify control flow and profiled data (Sec. III-B) account for the rest. Abstracting FP computations with subgraphs eliminates the need for human comprehension of underlying computations in the form of symbolic expressions (which is often impossible in large-scale applications). Poseidon's subgraph extraction phase is fully extensible to other FP precisions and functions.

D. Candidate Generation

Poseidon automatically generates two kinds of candidate rewrites for FP subgraphs.

Algebraic Rewrites. Poseidon automatically calls external tools to generate algebraic rewrites. This avoids manual effort such as writing symbolic expressions, preparing tool-specific DSL inputs, and converting results back into compiler IR. Poseidon scans FP subgraphs to build symbolic expressions, annotates it with profiled information, and calls registered rewriting tools. It then parses returned rewrites and constructs AST nodes. Poseidon can be extended to use new rewriting tools and algorithms through registration.

Precision Changes. Poseidon proposes precision changes to FP subgraphs. Since the configuration space is exponential in subgraph sizes, Poseidon uses ADAPT-style sensitivity metrics (Sec. III-B) to generate a constant number of rewrites: it ranks instructions by profiled sensitivities, prioritizes reductions on less sensitive ones, and emits candidate rewrites at configurable sensitivity percentiles. Supporting other data types or alternative tuning algorithms requires only proper registration.

E. Candidate Evaluation

Poseidon automatically predicts the performance/accuracy impacts of candidate rewrites with internal cost and accuracy models, which removes the need for human expertise in quantitative error analysis.

Cost Model. Poseidon materializes FP rewrites to temporary functions and runs standard optimization passes on the changed functions. Optimization passes like CSE give Poseidon better understanding of the actual performance implications of rewrites. For example, evaluating $\sqrt{t} + \sin(t)$, where t is itself a complex expression, only requires evaluating t once. After optimizations, Poseidon computes weighted sums of per-instruction costs of FP subgraphs φ:

$$\textsc{Cost}(\varphi) = \sum_{I \in \varphi} \textsc{InstCost}(I) \cdot \textsc{ExecutionCount}(I)$$

where the execution counts come from profiled data. Poseidon supports two cost models for determining per-instruction cost: a simple weighted instruction count and a custom cost model based on just-in-time (JIT) compilation of LLVM instructions. We present an empirical comparison of these cost models in Sec. V-A. New cost models can be registered directly in the Candidate Evaluation phase.

Weighted Instruction Count. The first cost model estimates the runtime cost by assigning each instruction a fixed count of expected cycles, as determined by LLVM's target-specific utility `TargetTransformInfo`. For example, basic FP operations like addition are assigned one cycle. Math library calls like `sin` are assigned four or more cycles.

Instruction-JIT. Another cost model is obtained by just-in-time (JIT) compiling microbenchmarks of FP instructions and measuring their runtimes (rounded to nearest integers).

For a FP subgraph φ, Poseidon estimates the cost contribution of a candidate rewrite $\mathcal{T}$ as follows:

$$\Delta\textsc{Cost}(\mathcal{T}) = \textsc{Cost}(\mathcal{T}\varphi) - \textsc{ErasableCost}(\varphi)$$

where $\mathcal{T}\varphi$ represents the changed subgraph. Because intermediate instructions may be external uses and are thus non-erasable, Poseidon only subtracts the cost of those instructions that become erasable as a result of the rewrite.

Accuracy Model. Poseidon evaluates the numerical accuracy of an FP subgraph φ by estimating its error contribution to the final result of the program. Specifically, Poseidon computes local errors at each output instruction o and weights them by aggregated gradients from FP profiles:

$$\textsc{Error}(\varphi) = \sum_{o \in \varphi} \textsc{LocalError}(\textsc{o}) \cdot \textsc{Gradient}(o)$$

To estimate local errors, Poseidon uniformly samples inputs within the hyperrectangle of profiled input bounds and computes both native FP results and MPFR [45] reference results. Following [21], Poseidon's MPFR evaluator iteratively increases precision until the leading mantissa bits converge. Finally, Poseidon aggregates the differences between native and reference results to produce local error estimates.

Given the error contribution estimation of an FP subgraph φ, Poseidon estimates the global error contribution of a candidate rewrite $\mathcal{T}$ as follows:

$$\Delta\textsc{Error}(\mathcal{T}) = \textsc{Error}(\mathcal{T}\varphi) - \textsc{Error}(\varphi)$$

where $\mathcal{T}\varphi$ represents the changed subgraph.

F. Solver

Poseidon uses a DP solver to find the rewrite set ϕ^* that minimizes the total error contributions:

$$\phi^* = \operatorname*{argmin}_{\phi} \sum_{\mathcal{T} \in \phi} \Delta\textsc{Error}(\mathcal{T})$$

subject to the constraint that total cost contributions do not exceed an internal cost budget:

$$\sum_{\mathcal{T} \in \phi} \Delta\textsc{Cost}(\mathcal{T}) \leq \textsc{CostBudget}$$

We present the solver algorithm in Algorithm 1. Poseidon processes evaluated rewrites for discovered expressions and subgraphs and populates a cost-to-error map, which maintains the minimum total error contribution achieved at each attainable total cost contribution, and a cost-to-solution map, which maintains the optimal rewrite set at each attainable cost. As inner loop iterations overwrite `c2e'` entries for colliding `c'`, this algorithm guarantees that at most one candidate rewrite can be applied for each expression or subgraph.

Since algebraic rewrites replace output instructions and thus alter the global error contributions of FP subgraphs, Algorithm 1 first processes all FP expressions, then updates the cost and error estimates of candidate rewrites for FP subgraphs (i.e., precision tuning) by excluding rewritten instructions. This adjustment improves solver accuracy as it excludes fictitious cost reductions from optimizing replaced instructions. During the solve, optional pruning steps remove any entry that leads to a higher total error contribution than another entry with strictly lower cost. Given resulting maps, Poseidon extracts the optimal rewrite set that minimizes the global cost contribution subject to a user-specified error tolerance, or a rewrite set

```
c2e ← {0 : 0}
c2s ← {0 : ∅}
for each Expression/Subgraph s:
    c2e' ← c2e
    c2s' ← c2s
    for each (c, e) ∈ c2e:
        for each Candidate t of s:
            if ΔCOST(t) ≥ 0 ∧ ΔERROR(t) ≥ 0:
                continue
            c' ← c + ΔCOST(t)
            e' ← e + ΔERROR(t)
            if c' ∉ c2e'.KEYS or e' < c2e'[c']:
                c2e'[c'] ← e'
                c2s'[c'] ← c2s[c'] ∪ {t}
    PRUNE(c2e', c2s')
    c2e ← c2e'
    c2s ← c2s'
return c2s
```

Algorithm 1: Poseidon's DP solver processes evaluated rewrites to build cost-to-error (c2e) and cost-to-solution (c2s) maps with an optional pruning step.

that minimizes the global error contribution subject to a user-specified internal cost budget (see Sec. III-I).

The time complexity of this DP algorithm is $O(NT)$, where N is the maximum number of candidate rewrites (top precision changes and algebraic rewrites) per subgraph, and T is the number of distinct subgraph execution times. A user can set N to an arbitrary constant. T is bounded by the product of the instruction count and the number of distinct instruction execution times. As we bin the execution time per instruction to an integer cycle count, there is a constant number of distinct execution times. This algorithm is thus linear in the number of instructions, albeit with potentially large constants. One can trade off optimization time and solver accuracy by changing the granularity of the binning (e.g., rounding to the nearest 5 cycles), as well as limiting subgraph sizes.

G. Post-Optimizations

After materializing rewrites, Poseidon runs standard compiler analyses and optimizations (see Sec. IV) like CSE and DCE to further improve code size and runtime performance.

H. Implementation

Poseidon is implemented on top of Enzyme [46, 47, 48], a high-performance automatic differentiation (AD) framework for LLVM [49] and MLIR [50] programs. The current implementation of Poseidon targets programs expressed in LLVM IR. Poseidon's profiling pass is implemented as a special mode of Enzyme's reverse-mode AD that inserts profiler calls to record runtime values and gradients; compiling in this mode and linking against a profiler static library shipped with Poseidon produces an instrumented executable that emits FP profiles. Poseidon's optimization pass is an LLVM function pass that consumes the generated FP profiles. In the candidate generation phase, the current implementation uses Herbie [21] to generate algebraic rewrites for extracted symbolic expressions.

I. Usage

Fig. 7 illustrates Poseidon's PGO-like workflow. One would first mark functions to be optimized with `__enzyme_fp_optimize`. For pointer parameters, one would pass correspondingly shadow memory and seed memory locations the function writes to as outputs with nonzero values; relative magnitudes of gradient seeds can encode output importance (see Sec. V-C for example). One would then run the first compilation to generate an instrumented program and execute it to generate FP profiles. One would then recompile with the FP profiles to perform context-sensitive FP optimizations. Users may specify a relative error tolerance attribute (`enzyme_err_tol`) at the call site so that Poseidon selects FP rewrites that result in the most performant program satisfying this error bound. An additional fine-tuning mode of Poseidon accepts an internal cost index from the internal DP table to materialize a specific point on the computed performance/accuracy frontier. The first optimization run performs external tool calls and a full DP solve, whereas subsequent optimization runs reuse the cached table and apply FP rewrites directly. Poseidon's fine-tuning mode enables efficient exploration of the computed performance/accuracy frontier with small surrogate inputs.

IV. Interoperability

Poseidon interacts with various other compiler analyses and optimizations. Standard passes like memory optimizations and common subexpression elimination before and after Poseidon's profiling and optimization passes improve Poseidon's analyses and context-sensitive FP optimizations.

A. Memory Optimizations

Memory optimizations help identify larger FP subgraphs in subgraph extraction (Sec. III-C). Memory optimizations like `mem2reg` and scalar replacement of aggregates (SROA) promote memory accesses into SSA registers, enabling surrounding FP operations to be grouped into the same subgraphs. Without memory optimizations, intermediate results are often spilled to and reloaded from memory, which limits Poseidon's understanding of the numerical context.

B. Common Subexpression Elimination

Common subexpression elimination (CSE) improves Poseidon's subgraph extraction and candidate generation as it merges common subexpressions into the same subgraph, which improves Poseidon's understanding of the numerical context. Running CSE passes after materializing FP rewrites can further simplify computation, e.g., merging shared FP casts and materialized algebraic rewrites.

```
// In `foo.cpp`:
void func(double* x, double* y);  // To optimize
void func_opt(double* x,  double* y,
              double* sx, double* sy) {
  __enzyme_fp_optimize(func,
    x, y,    // Input parameters
    sx, sy,  // Shadow memory (importance weights)
    enzyme_err_tol, 1e-3  // Optional error tolerance
  );
}
```

```
# 1) Profiling Pass
clang++ -fplugin=ClangEnzyme.so -fpprofile-generate \
    foo.cpp -o foo_instrumented.exe -lEnzymeFPProfiler

# 2) Generate FP Profile
ENZYME_FP_PROFILE_DIR=./fpprofile ./foo_instrumented.exe

# 3) Optimization Pass
clang++ -fplugin=ClangEnzyme.so -fpprofile-use=./fpprofile \
    foo.cpp -o foo_opt.exe
```

Fig. 7: Using Poseidon to perform end-to-end FP optimizations. ***Left:*** Mark function for context-sensitive FP rewrites; shadow memory holds gradient seeds (higher = preserve/improve accuracy, lower = approximate). An optional error tolerance attribute enables automatic tradeoff selection. ***Right:*** Poseidon's PGO-like workflow: 1) compile an instrumented program with `-fpprofile-generate`, 2) run it to generate FP profiles, 3) recompile with `-fpprofile-use` to perform FP optimizations.

C. Vectorization

Poseidon can optimize vectorized computations at multiple granularities. By default, it analyzes the numerical context of each vector element independently and may apply multiple rewrites within a vector expression to maximize performance/accuracy benefits. Alternatively, one can configure Poseidon to apply a single rewrite per vector expression. Performing vectorization after the optimization pass further reduces code size and improves runtime performance.

D. Other Optimizations

`InstCombine` and `SimplifyCFG` further simplify computation and control flow, which lead to better subgraph extraction and runtime performance. We run these passes before and after Poseidon's optimization pass.

E. Fast-Math Optimizations

Poseidon optimizes programs with fast-math optimizations enabled. While developers usually exercise caution regarding fast-math's impact on numerical accuracy, it can lead to both accuracy and performance improvements when used with caution. For example in Fig. 3, fast-math enables optimizations that replace branches with `select` instructions, allowing Poseidon to extract larger subgraphs. In Sec. V, we present how Poseidon discovers outsized performance/accuracy benefits with fast-math optimizations enabled.

V. Evaluation

We evaluate Poseidon on multiple case studies. To confirm the structured critical value hypothesis and emulate a realistic profiling setting, we run Poseidon's profiling pass on small surrogate input sets and run the optimization pass to generate performance/accuracy tradeoffs. We quantify numerical accuracy using geomean and worst-case relative errors in program outputs. We construct MPFR baseline programs with ForCE [51] (Sec. V-A) and RAPTOR [52] (Sec. V-B, V-C, V-D) to compute reference results.

We perform experiments on a 96-core AMD Ryzen Threadripper PRO 7995WX CPU with 512 GB of RAM. We build Poseidon with LLVM 21.0.0 and Herbie 2.3 on the host machine, and with LLVM 15.0.7 and Herbie 2.2 in STOKE's

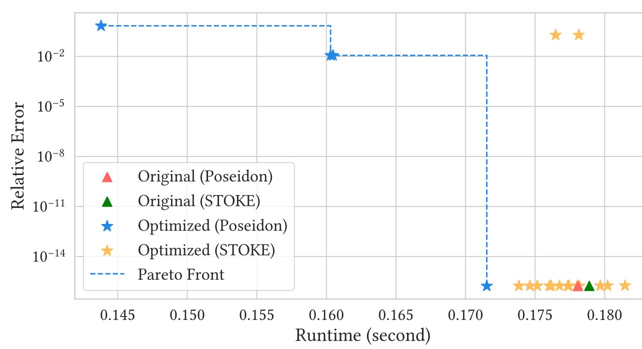

Fig. 8: Comparison of optimized `turbine1` programs from Poseidon and STOKE. Blue and yellow stars represent optimized programs from Poseidon and STOKE, respectively. The red and green triangles represent baseline programs for Poseidon and STOKE, respectively. The blue dashed line represents the Pareto front of optimized programs. *Lower-left is better.*

Docker container (see Sec.V-A). All programs are compiled with `-O3 -ffast-math -march=native` optimizations.

A. FPBench

We evaluate Poseidon on FPBench [53, 54], a suite of FP microbenchmarks written in FPCore. We translate these microbenchmarks to C for evaluation. For each benchmark, we run Poseidon's profiling pass on 100 input configurations randomly sampled from "preconditions" and test optimized programs on 100,000 different test cases. For comparison, we compute reference results using MPFR floats. Poseidon delivers performance/accuracy tradeoffs for 84 microbenchmarks (10 of them contain control flow) with default configurations. Poseidon enables accuracy improvements in 58% of these benchmarks, reducing relative errors by a geomean of $1.38\times$ (max: $3.46\times$), and a maximum speedup of $1.82\times$ with relative error of less than 10^{-6}.

Prior Work Comparison. We compare performance/accuracy tradeoffs found by Poseidon with optimized programs produced by STOKE [25], a stochastic optimizer for x86 binaries. Due to STOKE's limited support for math library functions and control flow constructs, we use both tools to

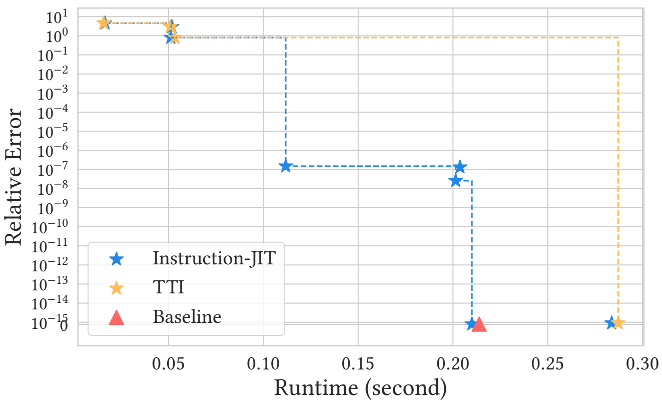

Fig. 9: Pareto fronts of optimized `logexp` programs using Poseidon's Instruction-JIT cost model (blue) and LLVM's `TargetTransformInfo` cost model (yellow). The red triangle represents the original program. Stars represent optimized programs. *Lower-left is better.*

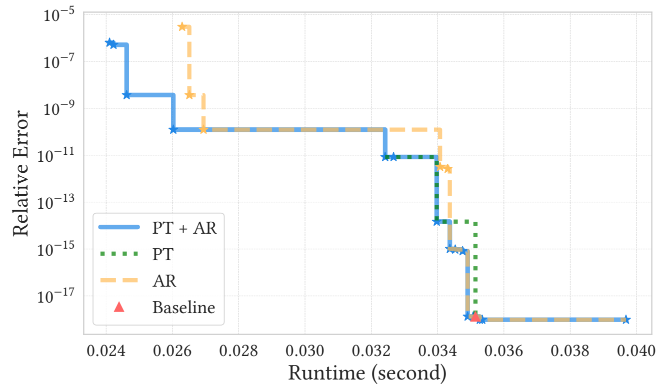

Fig. 10: Pareto fronts for `dquat`. Solid blue = Precision Tuning (PT) + Algebraic Rewriting (AR), dashed yellow = AR only, dotted green = PT only. Red triangle is the baseline; stars are optimized programs. *Lower-left is better.*

optimize microbenchmark `turbine1` [55, 56]. We compare 16 optimized programs from STOKE to all 4 performance/accuracy tradeoffs from Poseidon. As shown in Fig. 8, Poseidon delivers substantial performance benefits through approximations, whereas inaccurate alternatives from STOKE do not offer. Operating on LLVM IR allows Poseidon to target any instruction set architecture and fully support LIBM math functions. As a deterministic tool, Poseidon has the benefit of obtaining all optimization options in one run, whereas STOKE can require multiple expensive stochastic searches to discover useful alternatives, which inherently limits its scalability to practical applications.

Cost Model Comparison. We evaluate Poseidon's Instruction-JIT cost model with LLVM's `TargetTransformInfo` (TTI) cost model on the `logexp` benchmark. As Fig. 9 shows, Instruction-JIT supports a wider spectrum of performance/accuracy tradeoffs, including substantial speedups with geomean relative errors around 10^{-7} through precision changes. These FP rewrites are unavailable to the TTI cost model as it treats all FP precisions identically.

B. Quaternion Differentiator

We optimize `dquat`, a program that computes the derivative of quaternions with respect to input exponential maps. We use randomly generated small rotations, $\mathbf{v}$, with $||\mathbf{v}|| < 10^{-9}$, and larger rotations with $||\mathbf{v}|| \geq 10^{-9}$ as inputs, and compare relative errors (w.r.t. Frobenius norms) across output derivative matrices against MPFR reference. We generate FP profiles with 1,000 input matrices and evaluate optimized programs on 1,000,000 different matrices. `dquat` contains multiple loops that iterate through vector elements through memory accesses to compute norms and scalar products. To our knowledge, no prior work is capable of automatically rewriting numerical programs of this complexity. Poseidon leverages standard optimization passes like `mem2reg`, SROA, and loop unrolling to enable automatic extraction of l^2-norms expressions like $\theta = \sqrt{x^2+y^2+z^2}$ and Jacobian entry expressions like $u^2(0.5\cos(\theta/2) - \sin(\theta/2)/\theta) + \sin(\theta/2)/\theta$.

We present results in Fig. 10. Poseidon's DP solver effectively composes precision changes with algebraic rewrites to find multiple performance/accuracy tradeoffs that neither technique finds in isolation. For example, composing two techniques leads to a speedup of $1.46\times$ (upper left corner) with a geomean relative error of 6.23×10^{-7} (worst-case error: 0.067) by lowering nonlinear operations to FP32 and replacing $u^2(0.5\cos(\theta/2) - \sin(\theta/2)/\theta) + \sin(\theta/2)/\theta$ with $\sin(\theta/2)/\theta$ as profiled data reveal that $0 < \theta < 0.2$ and $u^2 < 1$ and thus $|u^2(0.5\cos(\theta/2) - \sin(\theta/2)/\theta)| < 0.002$, which is much smaller than $0.499 < \sin(\theta/2)/\theta < 0.5$. Poseidon also finds other tradeoffs that applies precision lowering on top of algebraic approximations without further downgrading accuracy, for example, a $1.35\times$ speedup at error 1.23×10^{-10} (worst-case error: 0.067). Poseidon is also capable of selecting 3 rewrites from Herbie that add special handling for small vector components, reducing geomean relative error by 26.8% (worst-case error -4.8%) with $1.13\times$ more compute time.

Artificially Distorted Bounds. We artificially scale the profiled bounds by a factor α, i.e., range $[c-w, c+w]$ becomes $[c-\alpha w, c+\alpha w]$, to see how worse profiled data affect optimization. Fig. 11 shows that substantially less accurate bounds with $\alpha > 10^8$ eliminates most numerical approximations with $10^{-14} \leq \epsilon \leq 10^{-7}$. Substantially more biased bounds with $\alpha < 10^{-4}$ reduce accuracy improvements. Given slightly distorted bounds with $0.1 < \alpha < 10$, Poseidon still produces a wide spectrum of optimization options. This experiment shows that FP profiles only need to be accurate within rough orders of magnitude to drive effective optimizations for `dquat`.

C. Eigensolver

We optimize a C++ program that solves the eigenvalue problem for 3×3 symmetric matrices with a robust algorithm [57]. Fig. 12 shows the most expensive steps that compute λ_1, which involve expensive FP operations such as fractional

TABLE I: Performance/accuracy tradeoffs discovered by Poseidon for a 3×3 eigensolver. Each Δ entry reports *geomean / worst-case* relative error over 100,000 test cases disjoint from the 10 profiling inputs (lower is better). The first two rows are hand-written F64 and F32 baselines. Speedups are relative to the F64 baseline. w values are gradient seeds used in the profiling pass. ✗ marks configurations with geomean error ≥ 1. Bold indicates improvements over the FP64 baseline.

Speedup	$\Delta\lambda_1$	$\Delta\lambda_2$	$\Delta\lambda_3$	Δx_1	Δx_2	Δx_3
1.00×	1.9e-16 / 7.2e-16	1.5e-16 / 4.2e-16	1.5e-16 / 4.2e-16	2.7e-16 / 1.0e-14	2.9e-16 / 5.4e-14	2.3e-16 / 5.4e-14
1.21×	4.3e-8 / 3.6e-7	1.8e-8 / 1.2e-7	2.0e-8 / 1.8e-7	1.4e-7 / 4.9e-6	1.7e-7 / 4.2e-5	1.3e-7 / 4.2e-5
	$w = 1$	$w = 1$	$w = 1$	$w = 1$	$w = 1$	$w = 1$
0.94×	**1.6e-16 / 5.6e-16**	1.5e-16 / 4.4e-16	1.5e-16 / 4.4e-16	**2.3e-16 / 5.8e-15**	3.3e-16 / 6.5e-14	2.9e-16 / 6.5e-14
1.19×	1.4e-8 / 1.4e-7	2.9e-8 / 2.1e-7	3.0e-8 / 2.7e-7	1.4e-7 / 4.9e-6	1.6e-7 / 3.9e-5	1.3e-7 / 3.9e-5
	$w = 1$	$w = 10^{-5}$	$w = 10^{-5}$	$w = 0$	$w = 0$	$w = 0$
1.12×	**1.7e-16 / 5.6e-16**	1.5e-16 / 4.4e-16	1.5e-16 / 4.4e-16	✗	✗	✗
1.55×	**1.7e-16 / 5.6e-16**	0.052 / 0.74	0.064 / 0.95	✗	✗	✗
2.11×	**4.3e-8 / 3.7e-7**	0.15 / 1.56	0.15 / 1.76	✗	✗	✗

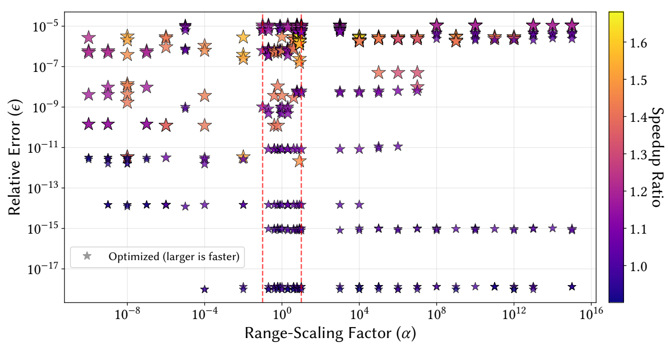

Fig. 11: Optimized `dquat` programs under varying range-scaling factors. Stars represent Pareto-optimal programs. Vertically aligned stars share the same factor. Lighter hues and larger stars indicate better runtime performance. Red dashed lines represent $\alpha = 0.1$ and $\alpha = 10$.

```
double t = (0.5 * J₃) * pow(3.0 / J₂, 1.5);
double α = acos(fmin(fmax(t, -1.0), 1.0)) / 3.0;
if (6.0 * α < π)
  λ₁ = 2.0 * sqrt(J₂ / 3.0) * cos(α);
else
  λ₁ = 2.0 * sqrt(J₂ / 3.0) * cos(α + 2.0 * π / 3.0);
```

Fig. 12: Steps that compute the first eigenvalue given the second and third invariants (J_2 and J_3) of a deviatoric matrix.

powers, (inverse) trigonometric functions, and square roots. Remaining computations build on λ_1 and consist of several additional dot/cross products and normalizations. 3 loops and 6 branches are in the eigensolve function, including a top-level guard on J_2 and a root-selection branch on α (see Fig. 12). This program takes a 3×3 symmetric matrix as input and produces three eigenpairs (λ_1, x_1), (λ_2, x_2), and (λ_3, x_3) as outputs. We profile this eigensolver under 10 input matrices $\mathbf{A} = \mathbf{I}_3 + \mathbf{S}^\top + \mathbf{S}$, where entries of $\mathbf{S}$ are uniformly sampled from $[-0.1, 0.1]$, and evaluate the optimized programs under 100,000 different matrices. We compute reference outputs using MPFR floats with 256 mantissa bits. Rows 1 and 2 in Table I show the accuracy statistics of the baseline programs.

We present a spectrum of performance/accuracy tradeoffs in Table I. Poseidon delivers accuracy improvements on the first eigenpair (row 3, worst-case error -33%) through context-sensitive term reassociations that mitigate roundoff errors, and a speedup comparable to the hand-written FP32 baseline through selective precision lowering (row 4). Biasing gradient seeds towards λ_1 (row 5-7) further prioritizes the accuracy of λ_1 and yields a $1.12\times$ speedup from approximating eigenvector computation (row 5). Additional approximations on λ_2 and λ_3 further improve program performance (row 6 and 7). These tradeoffs are discovered automatically by Poseidon's DP solver which selectively approximates nonlinear computations, such as vector normalization and characteristic polynomial solves, using low-order Taylor approximations.

Subgraph Extraction. Poseidon's subgraph extraction phase discovers a monolithic subgraph consisting of 29 input values and 49 output instructions with 201 intermediate FP instructions. Direct candidate generation on this monolithic subgraph is expensive due to its computational structure. For example, recursively expanding a "normalize $\rightarrow$ project $\rightarrow$ normalize $\rightarrow$ cross" chain in λ_1 evaluation until reaching input matrix entries can result in a 5,000-character symbolic expression[2]. To reduce optimization time, Poseidon can either skip rewriting excessively long expressions (note that they still benefit from rewrites applied to their subexpressions), or split a monolithic subgraph at high-fanout, sufficiently complex subexpressions to smaller subgraphs to avoid the generation of long expressions. However, splitting substantially weakens context-sensitive FP rewriting: splitting at α and normalization steps into three subgraphs halves end-to-end optimization time but eliminates the accuracy improvement (row 3) and the $1.19\times$ speedup (row 4). This is because splitting subgraphs reduces numerical context visible to candidate generation phase and introduces more non-erasable intermediates, both of which reduce opportunities for accuracy-improving rewrites and ap-

[2]With the batching feature in Herbie, evaluation of common subexpressions is simplified, but end-to-end optimization still takes 50 minutes on our setup.

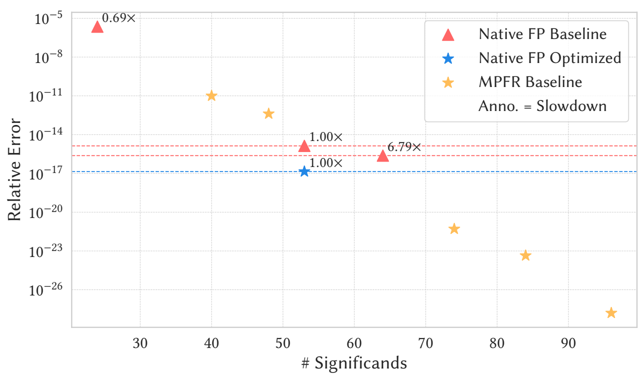

Fig. 13: Relative errors vs. number of significands used for size-50 LULESH runs (*lower is better*). The blue star marks the optimized FP64 program, yellow stars mark MPFR baselines, and red triangles mark native baselines with `float`, `double`, and `x86_fp80`, from left to right. Annotations show runtime ratios relative to the `double` baseline (*lower is better*).

proximations. As a key novelty, Poseidon interoperates with pre-optimizations (described in Sec. III-A) that coalesce FP instructions into larger FP subgraphs and thus enable rewrites that are not available when considering smaller subgraphs in isolation.

D. LULESH

LULESH [58] is an application from DOE's set of proxy exascale computing benchmarks that simulates Lagrangian hydrodynamics over a specified time window. It consists of approximately 5600 lines of C++ with over 200 `for` loops with runtime value-dependent `if` statements. MPFR reference results indicate that the original FP64 implementation gradually loses precision: for problem size 50, the computed final origin energy deviates from the reference FP64 result by 12 units in the last place (ULPs) after 1662 iterations. We generate a small surrogate profile from 100 iterations on problem size 30 ($\approx$ 1.3% of tested computation) to guide Poseidon's optimization pass.

Poseidon recovers the lost precision through context-sensitive FP rewrites. As shown in Fig. 13, 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. Poseidon achieves this with 9 rewrites that improve accuracy and 2 rewrites that simplify computation. This optimized program is more performant and accurate than an FP80 variant (with hardware-accelerated `x86_fp80`), which is 6.79× slower than the FP64 baseline but still incurs a ULP error of 2. While MPFR floats can produce more accurate results than our optimized program, running an MPFR implementation requires at least 500× more compute time than a native FP64 implementation. This result shows that Poseidon can recover the performance loss due to suboptimal choices of FP expressions within the program, achieving accuracy unachievable through precision changes at comparable cost. Poseidon completes an end-to-end run for an application of this scale within a reasonable amount of time: it takes 4 hours to complete external tool invocations and a full DP solve; subsequent compilations for the fine-tuning mode take only 0.3 second on average.

VI. Related Work

A. Mixed-Precision Tuning

ADAPT [27] offers a gradient-based error metric and a greedy mixed-precision allocation algorithm. Precimonious [18] searches for mixed-precision allocations under certain accuracy constraints. FPTuner [17] generates mixed-precision allocations from expression specifications and error thresholds. Unlike these tools which require manual code edits or DSL inputs, Poseidon fully automates the synthesis, evaluation, and application of precision changes. One can conveniently extend Poseidon to support more precision types and external mixed-precision allocation tools.

B. Expression Rewrites

Herbie [21, 59] automatically discovers rewrites that improve numerical accuracy of FPCore programs. Daisy [22] searches for algebraic rewrites to optimize straight-line Scala programs. Salsa [23] statically analyzes programs written in a C-like DSL and applies source-to-source transformations to improve their numerical accuracy. Unlike these tools which require DSL inputs to provide symbolic expressions and numerical context, Poseidon instruments programs to capture numerical context and automatically constructs DSL inputs. To our knowledge, Poseidon is the first framework that performs algebraic rewrites at a full application scale.

C. Profile-Guided Optimization

Modern compilers profile user programs to collect runtime information for advanced compiler optimizations [60]. Several existing works use profiling techniques to collect numerical context required in FP optimization: Precimonious [18] dynamically profiles mixed-precision configurations and Herbie [21] profiles algebraic expressions. Poseidon uses a profiling pass within the compiler to automatically collect numerical context needed to ensure the effectiveness of FP rewrites through small surrogate profiling runs.

VII. Conclusion

We have presented an end-to-end framework, Poseidon, that automates context-sensitive floating-point optimizations within a production compiler. Through its PGO-like two-phase compilations and interoperability with standard compiler analyses and optimizations, Poseidon removes the need for human expertise of numerical methods and understanding of specific numerical context required to perform correct and performant rewrites. We have evaluated Poseidon on multiple large-scale applications with small surrogate profiling runs and performed ablations on multiple components of its design. Our results have demonstrated that even with small surrogate profiling data, Poseidon automatically delivers outsized performance/accuracy benefits in large-scale applications.

ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation under Grant No. 2346519. This work was also supported in part by the Alan Turing Institute, and the U.S. Department of Energy. We gratefully acknowledge the support of the NSF-Simons AI-Institute for the Sky (SkAI) via grants NSF AST-2421845 and Simons Foundation MPS-AI-00010513. This work was also supported by JST SPRING, Japan Grant Number JPMJSP2106, JPMJSP2180, and the RIKEN Junior Research Associate Program. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Assistant Secretary for Environmental Management, Office of Science, under contract DE-AC02-06CH11357. We thank the anonymous reviewers for their valuable feedback.

DATA-AVAILABILITY STATEMENT

The reproduction artifact of this paper is available on Zenodo [61]. The latest version of this artifact is available on GitHub: `https://github.com/PRONTOLab/Poseidon`.

APPENDIX

A. Abstract

We provide an artifact to help reproduce this paper's main experimental results (Figures 9, 10, 13, and Table I).

B. Artifact check-list (meta-information)

- **Program:** Enzyme; LLVM; Herbie; Python 3; Racket (all included in the provided Docker image).
- **Docker Image:** `sbrantq/poseidon:latest`.
- **Run-time environment:** x86-64 Linux; Docker.
- **Hardware:** x86-64 machine; at least 8 CPU cores; at least 16 GB of RAM.
- **Execution:** Run the provided Docker container and execute the included scripts to reproduce three figures and one table.
- **Output:** Figures 9, 10, 13, and Table I.
- **How much disk space required (approximately)?:** 15 GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 1 hour.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Archived (provide DOI)?:** `10.5281/zenodo.18004991`

C. Description

1) How delivered: This artifact is available at `https://github.com/PRONTOLab/Poseidon` and is archived at `https://doi.org/10.5281/zenodo.18004991`.

2) Hardware dependencies: No hardware dependencies.

3) Software dependencies: Docker.

D. Installation

1) **Docker Image**
   a) Install Docker on the host machine.
   b) Start the container:

```
sudo docker run -it sbrantq/poseidon:latest \
  /bin/bash
```

Note: Results are hardware-dependent. The Docker image includes a cost model generated on our machine (AMD Ryzen Threadripper PRO 7995WX). For best performance and reproducibility on different hardware, we recommend building from source and regenerating the cost model (see Sec. 2).

2) **Build From Source**
   a) **Prerequisites:**

```
sudo apt install build-essential cmake \
  ninja-build libmpfr-dev
pip install lit numpy matplotlib tqdm
```

Additionally, install Racket (`https://racket-lang.org/`) and Rust (`https://www.rust-lang.org/tools/install`).

   b) **Clone and Initialize Submodules:**

```
git clone https://github.com/PRONTOLab/Poseidon.git
cd Poseidon
git submodule update --init llvm-project Enzyme
```

   c) **Build LLVM:**

```
cd llvm-project
mkdir build && cd build
cmake -G Ninja \
  -DLLVM_ENABLE_PROJECTS="clang" \
  -DLLVM_ENABLE_LLD=ON \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DCMAKE_BUILD_TYPE=Release \
  ../llvm
ninja
cd ../..
```

   d) **Build Enzyme with Poseidon Enabled:**

```
cd Enzyme
mkdir build && cd build
cmake -G Ninja ../enzyme/ \
  -DLLVM_DIR=<...> \
  -DLLVM_EXTERNAL_LIT=$(which lit) \
  -DCMAKE_BUILD_TYPE=Release \
  -DENABLE_POSEIDON=ON \
  -DCMAKE_C_COMPILER=<...> \
  -DCMAKE_CXX_COMPILER=<...>
ninja
cd ../..
```

   e) **Regenerating the Cost Model:** The cost model (`cost-model/cm.csv`) is hardware-specific. To regenerate it for your machine:

```
cd $HOME/Poseidon/cost-model
python3 microbm.py
cp results.csv cm.csv
```

E. Experiment workflow

1) **FPBench cost model ablation** (Figure 9) can be reproduced with:

```
cd $HOME/Poseidon/FPBench/ablations
python3 ablation.py
```

Output: `plots/fptaylor-extra-ex11-ablation.png`.

2) **Quaternion differentiator results** (Figure 10) can be reproduced with:

```
cd $HOME/Poseidon/dquat
python3 run_ablation.py
```

Output: `dquat.png`.

3) **Eigensolver results** (Table I) can be reproduced with:

```
cd $HOME/Poseidon/eig
python3 run_cases.py
```

Outputs: `biased.txt` and `equal.txt` (see `eig/README.md` for details on how to interpret the output).

4) **LULESH results** (Figure 13) can be reproduced with:

```
cd $HOME/Poseidon/lulesh
python3 ablation.py
```

Output: `lulesh.png`.
Note: The 0-ULP result is hardware-dependent. To find the optimal configuration for your hardware, first regenerate the cost model, then run `make && python3 run.py && python3 benchmark.py –sample-percent 10`.

F. *Evaluation and expected result*

This artifact should reproduce the following results:

- Figure 9, saved as `FPBench/ablations/plots/fptaylor-extra-ex11-ablation.png`.
- Figure 10, saved as `dquat/dquat.png`.
- Figure 13, saved as `lulesh/lulesh.png`.
- Table I entries in `eig/biased.txt` and `eig/equal.txt`.

Numerical differences can occur due to differences in the environment, but the general trends (e.g., performance/accuracy tradeoffs) should match the results in this paper.

G. *Reusing This Artifact*

This section describes how to apply Poseidon to a new benchmark.

1) **Set Up Paths.** Configure the paths to custom builds of LLVM and Enzyme:

```
export CLANG_PATH=<...>/llvm-project/build/bin
export ENZYME_PATH=\
  <...>/Enzyme/build/Enzyme/ClangEnzyme-X.so
export PROFILER_PATH=<...>/Enzyme/build/Enzyme
```

2) **Profiling Pass.** First, compile your program with floating-point profiling enabled to collect runtime information:

```
$CLANG_PATH/clang++ -O3 -ffast-math -march=native \
    -fplugin=$ENZYME_PATH \
    -mllvm --fpprofile-generate \
    -L $PROFILER_PATH -lEnzymeFPProfile \
    your_program.cc -o your_program_prof
```

3) **Generate Floating-Point Profiles.** Run the profiled executable with (small surrogate) inputs to generate floating-point profiles:

```
./your_program_prof <your_arguments>
```

This creates an `fpprofile` directory.

4) **Optimization Pass.** Compile with Poseidon's optimization pass enabled:

```
$CLANG_PATH/clang++ -O3 -ffast-math -march=native \
    -fplugin=$ENZYME_PATH \
    -mllvm --fpprofile-use=./fpprofile \
    -mllvm --fpopt-cost-model-path=\
        $HOME/Poseidon/cost-model/cm.csv \
    your_program.cc -o your_program_opt
```

This produces an optimized program (`your_program_opt`) that aims to improve accuracy while preserving performance. The first run invokes external tools and performs a full dynamic-programming solve, with results cached (in the `cache` directory by default). Subsequent runs reuse these cached results.

5) **(Optional) Generate and Evaluate Other Optimized Programs.** The first compilation generates `cache/budgets.txt` containing all achievable cost budgets from the dynamic-programming solve. To explore other performance/accuracy trade-offs:
   a) **Compile with different budgets:** Recompile with varying `--fpopt-comp-cost-budget` values from `cache/budgets.txt`. Each budget produces a differently optimized binary.
   b) **Benchmark:** Run each binary and compare outputs against a reference (e.g., the original program) to evaluate its performance and accuracy.

See `lulesh/run.py` and `lulesh/benchmark.py` for an example of automating this process.

REFERENCES

[1] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[2] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, pp. 583 – 589, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235959867

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 06 1953. [Online]. Available: https://doi.org/10.1063/1.1699114

[4] R. Car and M. Parrinello, "Unified approach for molecular dynamics and density-functional theory," *Phys. Rev. Lett.*, vol. 55, pp. 2471–2474, Nov 1985. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.55.2471

[5] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.*, vol. 140, pp. A1133–A1138, Nov 1965. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.140.A1133

[6] A. D. Becke, "Density-functional thermochemistry. iii. the role of exact exchange," *The Journal of Chemical Physics*, vol. 98, no. 7, pp. 5648–5652, 04 1993. [Online]. Available: https://doi.org/10.1063/1.464913

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:205001834

[8] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[9] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.

[10] O. Abril-Pla, V. Andreani, C. Carroll, L. Dong, C. J. Fonnesbeck, M. Kochurov, R. Kumar, J. Lao, C. C. Luhmann, O. A. Martin *et al.*, "Pymc: a modern, and comprehensive probabilistic programming framework in python," *PeerJ Computer Science*, vol. 9, p. e1516, 2023.

[11] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of statistical software*, vol. 76, 2017.

[12] J. Osorio, A. Armejach, E. Petit, G. Henry, and M. Casas, "A bf16 fma is all you need for dnn training," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 3, pp. 1302–1314, 2022.

[13] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*. IEEE, 2019, pp. 36–39.

[14] S.-y. Liu, Z. Liu, X. Huang, P. Dong, and K.-T. Cheng, "Llm-fp4: 4-bit floating-point quantized transformers," *arXiv preprint arXiv:2310.16836*, 2023.

[15] S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei, "The era of 1-bit llms: All large language models are in 1.58 bits," *arXiv preprint*

arXiv:2402.17764, 2024.

[16] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT: Algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of SC '18*. Piscataway, NJ: IEEE Press, 2018, pp. 48:1–13.

[17] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846

[18] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503296

[19] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 333–343. [Online]. Available: https://doi.org/10.1145/3213846.3213862

[20] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "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*, ser. Lecture Notes in Computer Science, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., vol. 11501. Springer, 2019, pp. 227–246. [Online]. Available: https://doi.org/10.1007/978-3-030-20656-7_12

[21] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–11. [Online]. Available: https://doi.org/10.1145/2737924.2737959

[22] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '18. IEEE Press, 2018, p. 208–219. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00028

[23] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," in *Automated Formal Methods*, ser. Kalpa Publications in Computing, N. Shankar and B. Dutertre, Eds., vol. 5. EasyChair, 2018, pp. 63–76. [Online]. Available: /publications/paper/x58n

[24] I. Briggs, Y. Lad, and P. Panchekha, "Implementation and synthesis of math library functions," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632874

[25] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," *SIGPLAN Not.*, vol. 49, no. 6, p. 53–64, Jun. 2014. [Online]. Available: https://doi.org/10.1145/2666356.2594302

[26] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017. [Online]. Available: http://arxiv.org/abs/1710.03740

[27] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: Algorithmic differentiation applied to floating-point precision tuning," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 614–626.

[28] IEEE, "754-2019 - ieee standard for floating-point arithmetic," pp. 1–84, Jul. 2019.

[29] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres *et al.*, *Handbook of floating-point arithmetic*. Springer, 2018, vol. 1.

[30] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti, A. Daglis, N. E. Jerger, B. Falsafi, S. Misailovic, A. Sampson, and D. Zufferey, "Exploiting errors for efficiency: A survey from circuits to applications," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3394898

[31] C. Fang, T. Chen, and R. Rutenbar, "Floating-point error analysis based on affine arithmetic," in *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).*, vol. 2, 2003, pp. II–561.

[32] J. Laurel, S. B. Qian, G. Singh, and S. Misailovic, "Synthesizing precise static analyzers for automatic differentiation," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3622867

[33] M. Ogawa *et al.*, "Overflow and roundoff error analysis via model checking," in *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*. IEEE, 2009, pp. 105–114.

[34] P. Langlois, "A Revised Presentation of the CENA Method," INRIA, Tech. Rep. RR-4025, 2000.

[35] P. D. Hovland and J. Hückelheim, "Error estimation and correction using the forward cena method," in *Computational Science – ICCS 2021*, M. Paszynski, D. Kranzlmüller, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds. Cham: Springer International Publishing, 2021, pp. 765–778.

[36] N. J. Higham and T. Mary, "A new approach to probabilistic rounding error analysis," *SIAM journal on scientific computing*, vol. 41, no. 5, pp. A2815–A2835, 2019.

[37] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *Journal of Computational Science*, vol. 36, p. 101017, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750318309475

[38] N. J. Higham, "The accuracy of floating point summation," *SIAM Journal on Scientific Computing*, vol. 14, no. 4, pp. 783–799, July 1993.

[39] S. Gal, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software (TOMS)*, vol. 17, no. 1, pp. 26–45, 1991.

[40] M. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 628–637, 2000.

[41] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surv.*, vol. 53, no. 2, Apr. 2020. [Online]. Available: https://doi.org/10.1145/3381039

[42] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1074–1085. [Online]. Available: https://doi.org/10.1145/2884781.2884850

[43] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 333–343.

[Online]. Available: https://doi.org/10.1145/3213846.3213862

[44] H. Brunie, C. Iancu, K. Z. Ibrahim, P. Brisk, and B. Cook, "Tuning floating-point precision using dynamic program information and temporal locality," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[45] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, June 2007.

[46] W. Moses and V. Churavy, "Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 12 472–12 485. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

[47] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, "Reverse-mode automatic differentiation and optimization of gpu kernels via enzyme," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476165

[48] W. S. Moses, S. H. K. Narayanan, L. Paehler, V. Churavy, M. Schanen, J. Hückelheim, J. Doerfert, and P. Hovland, "Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[49] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[50] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: scaling compiler infrastructure for domain specific computation," in *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '21. IEEE Press, 2021, p. 2–14. [Online]. Available: https://doi.org/10.1109/CGO51591.2021.9370308

[51] P. D. Hovland and J. Hückelheim, "Error estimation and correction using the forward cena method," in *Computational Science – ICCS 2021*, M. Paszynski, D. Kranzlmüller, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, Eds. Cham: Springer International Publishing, 2021, pp. 765–778.

[52] F. Hoerold, I. R. Ivanov, A. Dhruv, W. S. Moses, A. Dubey, M. Wahib, and J. Domke, "Raptor: Practical numerical profiling of scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 661–680. [Online]. Available: https://doi.org/10.1145/3712285.3759810

[53] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a Standard Benchmark Format and Suite for Floating-Point Analysis," in *Numerical Software Verification - 9th International Workshop (NSV 2016)*, ser. Lecture Notes in Computer Science, vol. 10152, Toronto, Canada, Jul. 2016, pp. 63–77. [Online]. Available: https://hal.science/hal-03971234

[54] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock, *Combining Tools for Optimization and Analysis of Floating-Point Computations: 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, 07 2018, pp. 355–363.

[55] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," in *FM 2015: Formal Methods*, N. Bjørner and F. de Boer, Eds. Cham: Springer International Publishing, 2015, pp. 532–550.

[56] E. Darulova and V. Kuncak, "Sound compilation of reals," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 235–248. [Online]. Available: https://doi.org/10.1145/2535838.2535874

[57] W. Scherzinger and C. Dohrmann, "A robust algorithm for finding the eigenvalues and eigenvectors of 3×3 symmetric matrices," *Computer Methods in Applied Mechanics and Engineering*, vol. 197, no. 45, pp. 4007–4015, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782508001436

[58] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[59] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining precision tuning and rewriting," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, 2021, pp. 1–8.

[60] "Clang compiler user's manual: Profile guided optimization," https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization, LLVM Project, 2025, accessed: 2025-02-28.

[61] S. B. Qian, V. Sathia, I. R. Ivanov, J. Hückelheim, P. Hovland, and W. S. Moses, "Thinking fast and correct: Automated rewriting of numerical code through compiler augmentation," Dec. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.18004991