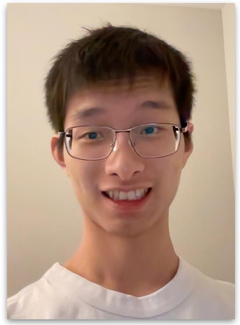
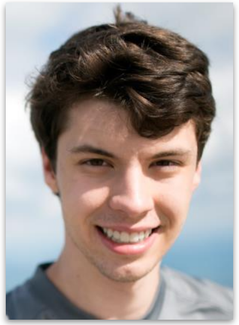
Poseidon: Profile-Guided Numerical Rewriting at Full-Application Scale





Siyuan Brant Qian[1] Vimarsh Sathia[1] Ivan R. Ivanov[3] Jan Hückelheim[2] Paul Hovland[2] William S. Moses[1]

siyuanq4@illinois.edu

[1] University of Illinois Urbana-Champaign, USA
[2] Argonne National Laboratory, USA
[3] Institute of Science Tokyo and RIKEN CCS, Japan

EGRAPHS 2026 @ PLDI, Boulder, CO
June 15, 2026

Optimizing numerical code is challenging at three levels:
Context, **Scope**, and **Scale**.
Profiling + **Compiler Optimization** are the key.

Context: Where You Change Matters

```
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

Context: Where You Change Matters

```java
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```java
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Activation:
1.6× speedup
error +≈ 1e-12 (safe!)

Context: Where You Change Matters

```java
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```java
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

```java
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Lower Activation:
1.6× speedup
error +≈ 1e-12 (safe!)

Lower Reduction:
error +≈ 1e-4 (risky!)

Context: Where You Change Matters

```java
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```java
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

```java
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Profiling just 10 iterations:

Reduction is $10^4 \times$ more sensitive*!

Lower Activation:
1.6× speedup
error +≈ 1e-12 (safe!)

Lower Reduction:
error +≈ 1e-4 (risky!)

* ADAPT metric (Menon et al. '18).

Context: Structured Critical Value Hypothesis

```java
double sum = 0.0;
for (double x : xs) {
    double val = sigmoid(exp(x));
    sum += val;
}
```

```java
double sum = 0.0;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

```java
float sum = 0.0f;
for (double x : xs) {
    float val = sigmoidf(expf(x));
    sum += val;
}
```

Profiling just 10 iterations:

Reduction is $10^4 \times$ more sensitive*!

Accuracy-critical values are often ***structural*** (e.g., reduction, cancellation, one-hot, thresholds). A **small surrogate profile** can reveal key information.

Scope: Precision Isn’t the Only Knob

FP32 (Original Order)

⚡ Fast ❌ Wrong

`(1e8 + 1) - 1e8 →  1.0000000e8 - 1e8 → 0`

+1 rounded away

Scope: Precision Isn’t the Only Knob

FP32 (Original Order)

⚡ Fast ❌ Wrong

`(1e8 + 1) - 1e8 → 1.0000000e8 - 1e8 → 0`

+1 rounded away

FP64 (Same Order)

🐢 Costly ✅ Correct

`(1e8 + 1) - 1e8 → 1.00000001e8 - 1e8 → 1`

Use FP64
(Tradeoff!)

Scope: Precision Isn’t the Only Knob

FP32 (Original Order)

`(1e8 + 1) - 1e8 → 1.0000000e8 - 1e8 → 0`

⚡ Fast ❌ Wrong

+1 rounded away

FP64 (Same Order)

`(1e8 + 1) - 1e8 → 1.00000001e8 - 1e8 → 1`

🐢 Costly ✅ Correct

Use FP64
(Tradeoff!)

FP32 (Reassociated)

`(1e8 - 1e8) + 1 → 0 + 1 → 1`

⚡ Fast ✅ Correct

Algebraic Rewrite
(circumvents tradeoff
but requires context)

Scale: The Math Is in the Mess

```
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

Math behind control flow,
memory I/O,
function calls, …;
existing source/binary-level
tools fail.

Scale: The Math Is in the Mess

```c
double foo(double x) {
  double y = 0.0;
  if (x > 0.0)
    y = pow(x, 3);
  return y;
}
```

```
define double @foo(double %x) {
entry:
  %p = call double @powi(
    double %x, i32 3)
  %r = call double @max(
    double 0.0, double %p)
  ret double %r
}
```

Math behind control flow,
memory I/O,
function calls, …;
existing source/binary-level
tools fail.

A production compiler makes math
explicit through optimizations
(`SimplifyCFG`, `mem2reg`, inlining, …)

The caller wants $max(0, x^3)$!

Can we **automate** numerical rewriting techniques in compilers?

How much performance are we **leaving on the table** due to suboptimal choices of precision and expressions?

Our Answer — Poseidon

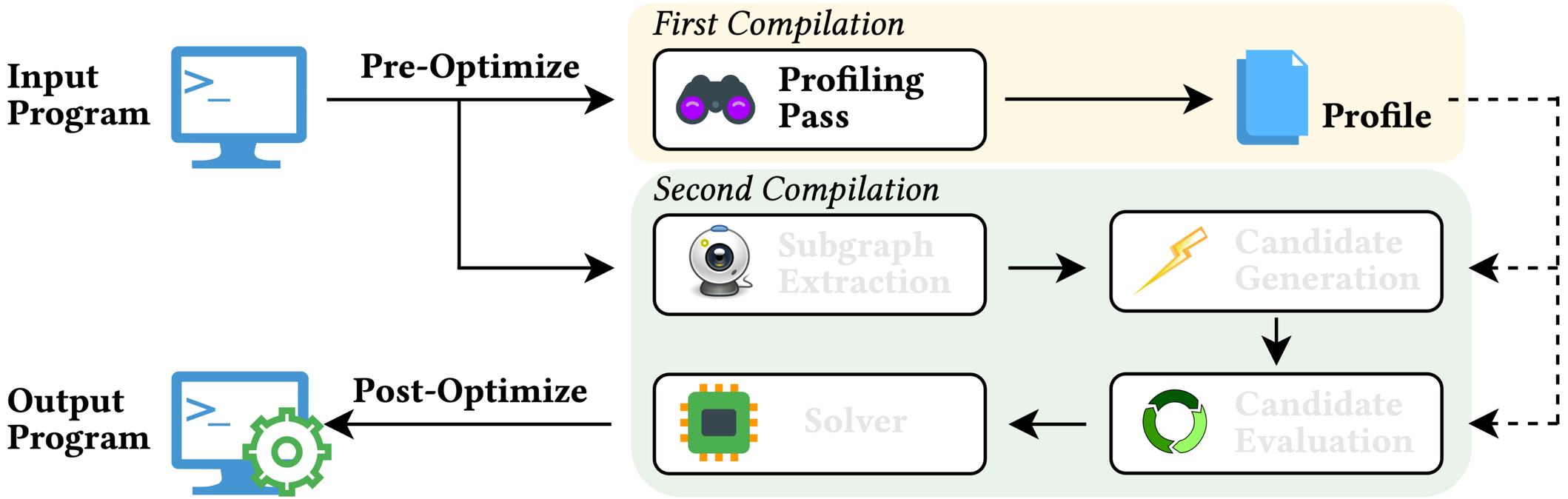
- The first framework that automates advanced numerical rewriting techniques within a production compiler

Our Answer — Poseidon

- The first framework that automates advanced numerical rewriting techniques within a production compiler
- **First Compile:** Instrument user program to collect numerical context

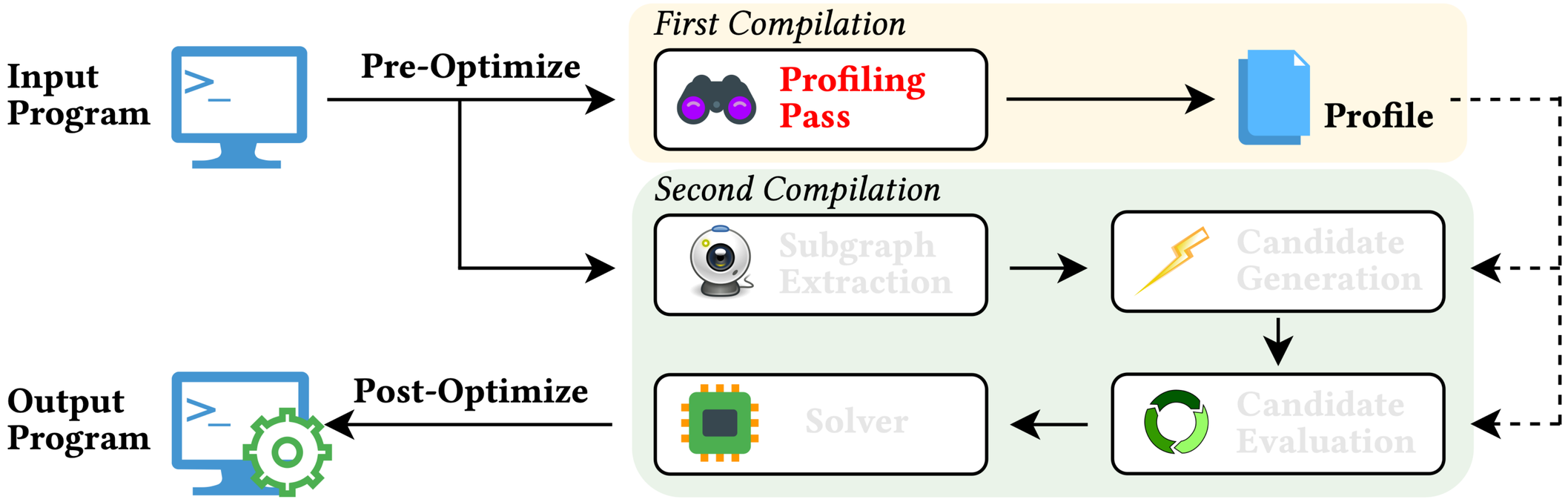
Our Answer — Poseidon

- The first framework that automates advanced numerical rewriting techniques within a production compiler
- **Second Compile:** Perform *full-application scale* numerical rewrites



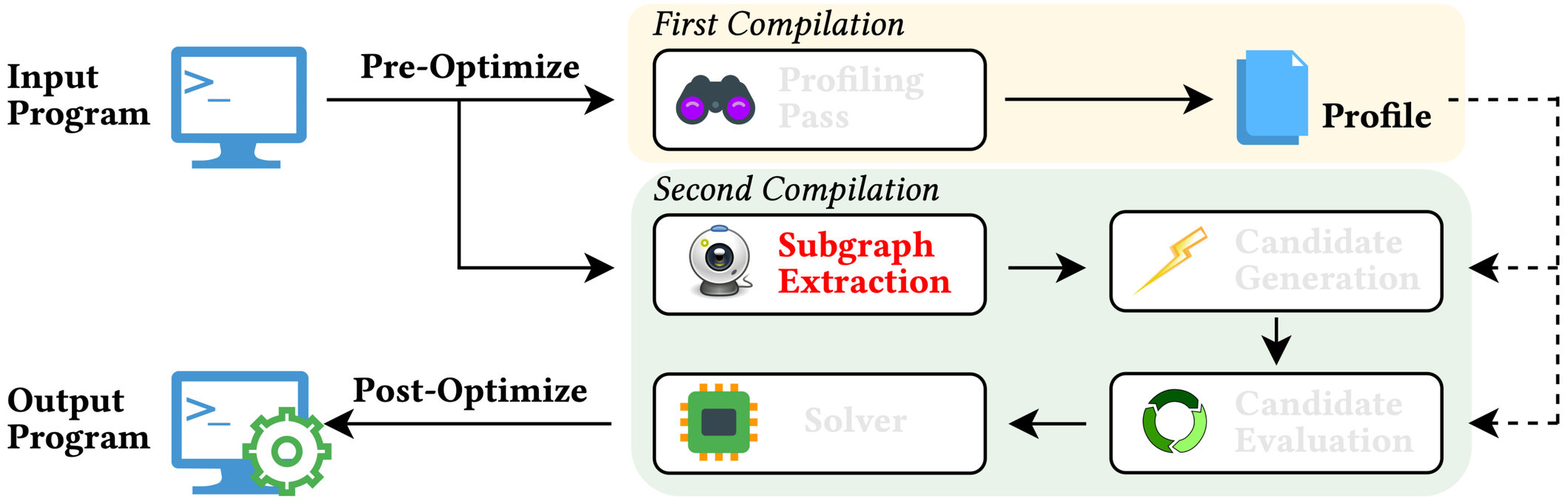
Profiling Pass

- Perform **pre-optimizations** (standard LLVM passes) to expose underlying math
- Emit profiles: execution counts, running sums of values/gradients per instruction



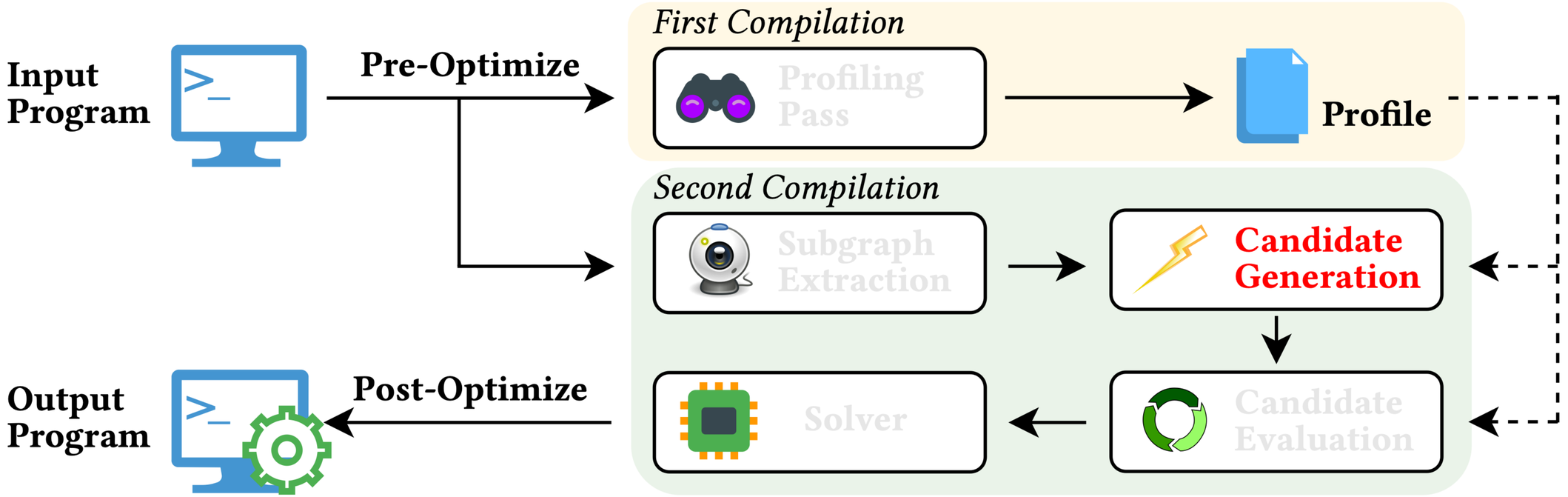
Subgraph Extraction

- Find expression graphs through flood-fill scan of LLVM IR
- Expression graphs are used to synthesize rewrites



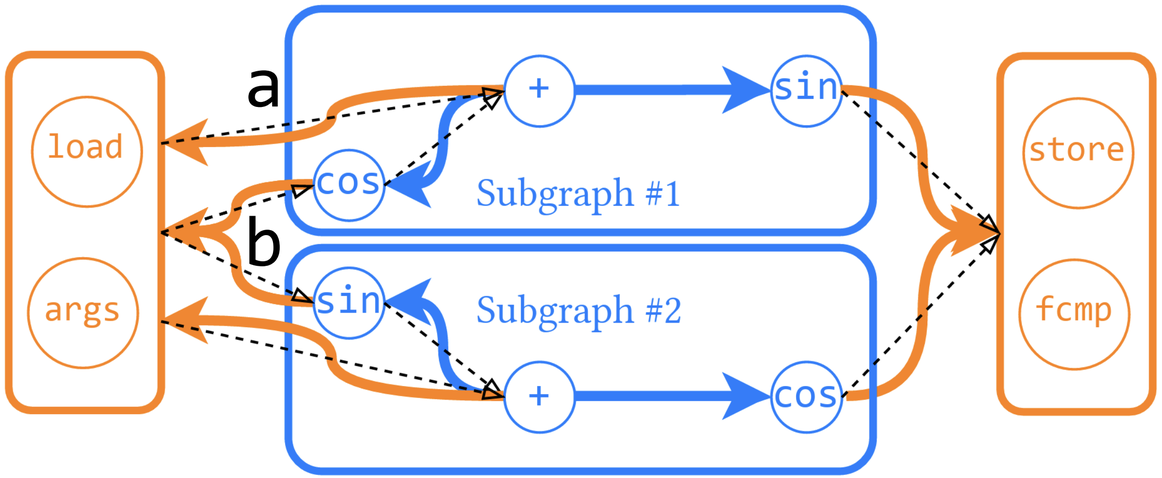
Candidate Generation

- Rewrite Class #1: algebraic rewrites (from Herbie)
- Rewrite Class #2: expression-graph precision changes
- Fully extensible to a broader scope of rewrites!



Algebraic Rewrites in Poseidon

- Poseidon invokes **Herbie**, a floating-point expression rewriting tools using equality saturation
- Poseidon lifts LLVM expression graphs with supported operations into Herbie inputs and embeds profiled bounds as “preconditions”; can be the compiler substrate for other tools



Profiled bounds → FPCore preconditions

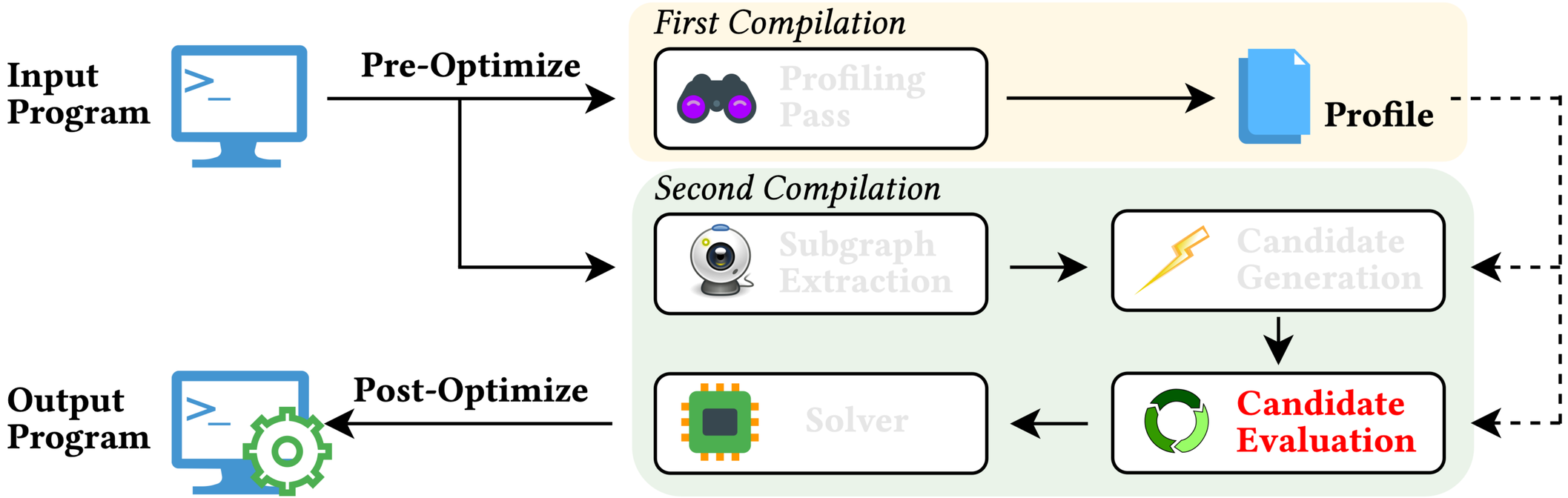
```
a ∈ [-3.14, 3.14]
b ∈ [0.0, 1.0e3]
```

Herbie Inputs:

```
(FPCore (a b)
:pre (and (<= -3.14 a 3.14)
          (<= 0.0 b 1.0e3))
  (sin (+ a (cos b))))
```

Candidate Evaluation

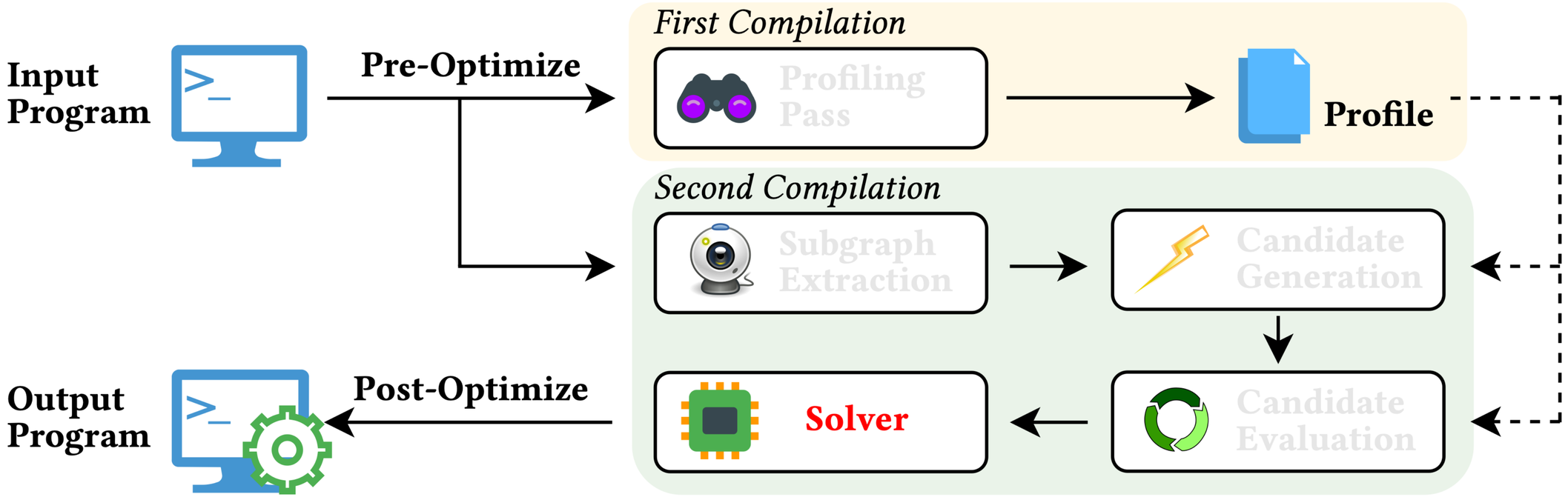
- Goal: predict candidate's performance/accuracy impact
- Cost: Sum of **cost × execution count** over affected instructions
- Accuracy: Sum over **local error × global sensitivity** over affected instructions



Solver (Global Selection of Rewrites)

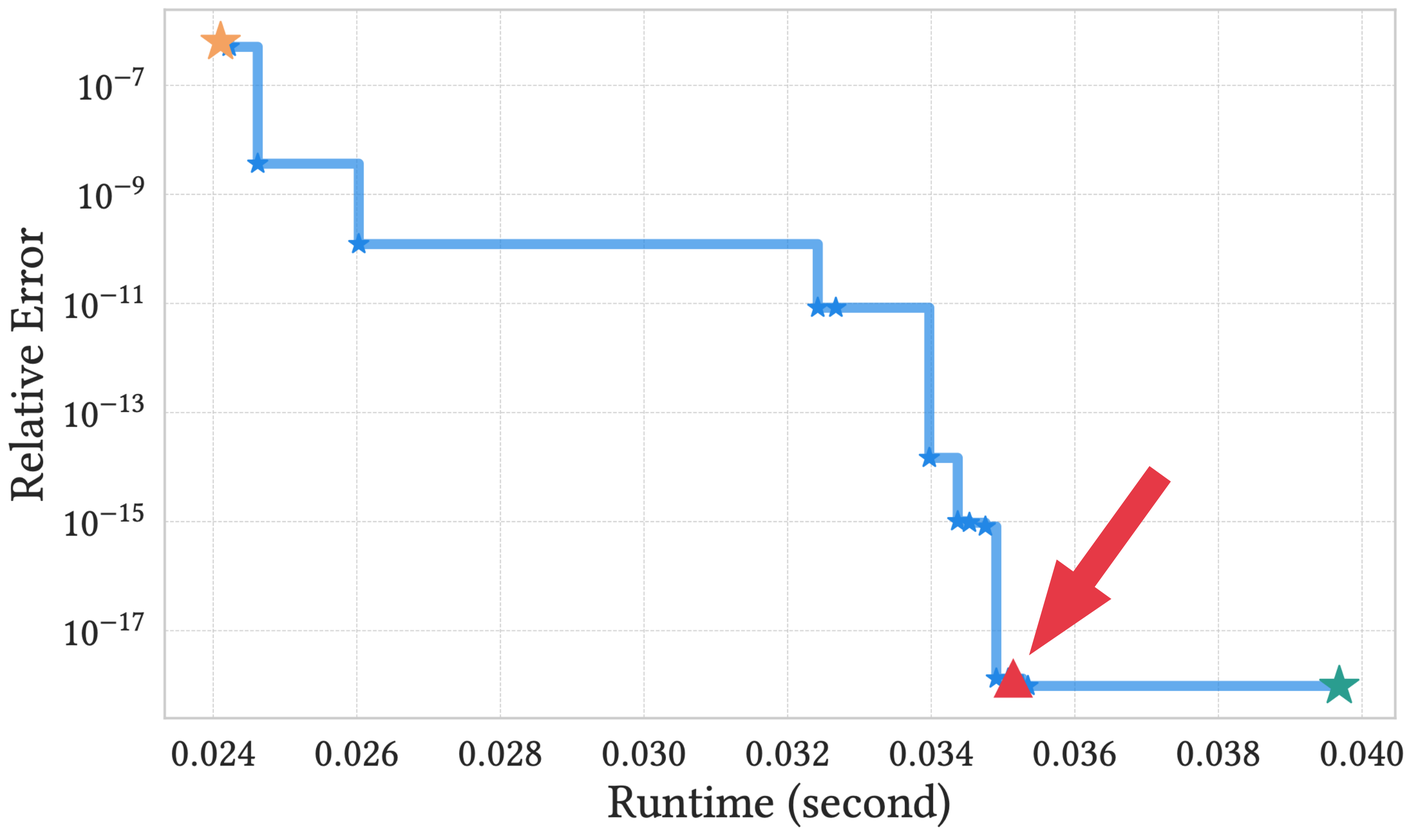
- Given a set of evaluated rewrites with estimated performance/accuracy cost, solve for a Pareto frontier with dynamic programming

- **Result: for a given cost/accuracy budget, Poseidon has the best rewrite strategy**



Evaluation: Quaternion Differentiator

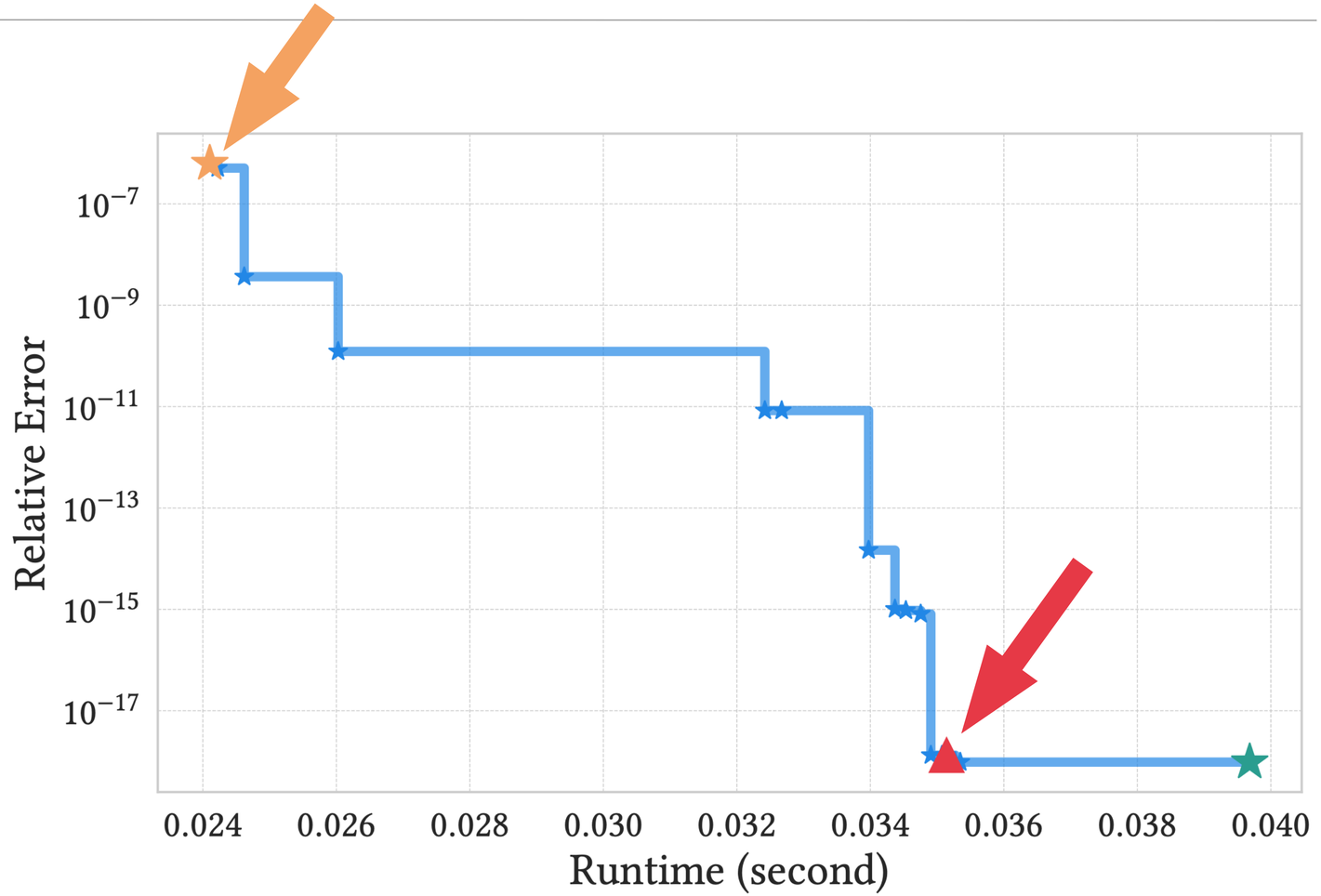
- Given the **original** program, Poseidon produces tradeoffs



Lower is more accurate; left is faster.

Evaluation: Quaternion Differentiator

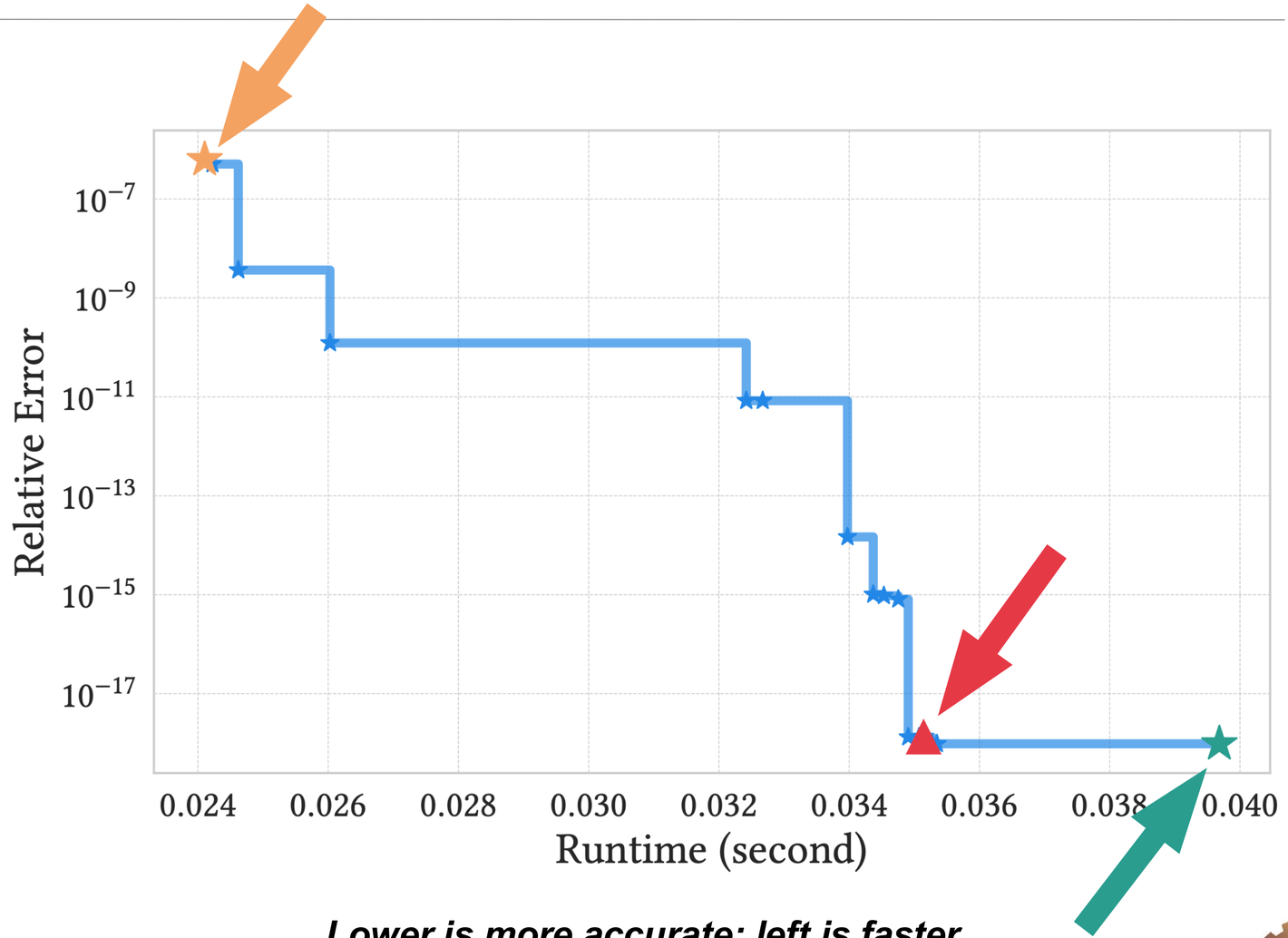
- Given the **original** program, Poseidon produces tradeoffs
- Trade accuracy for performance
  - **1.46×** with error of 6e-7



Lower is more accurate; left is faster.

Evaluation: Quaternion Differentiator

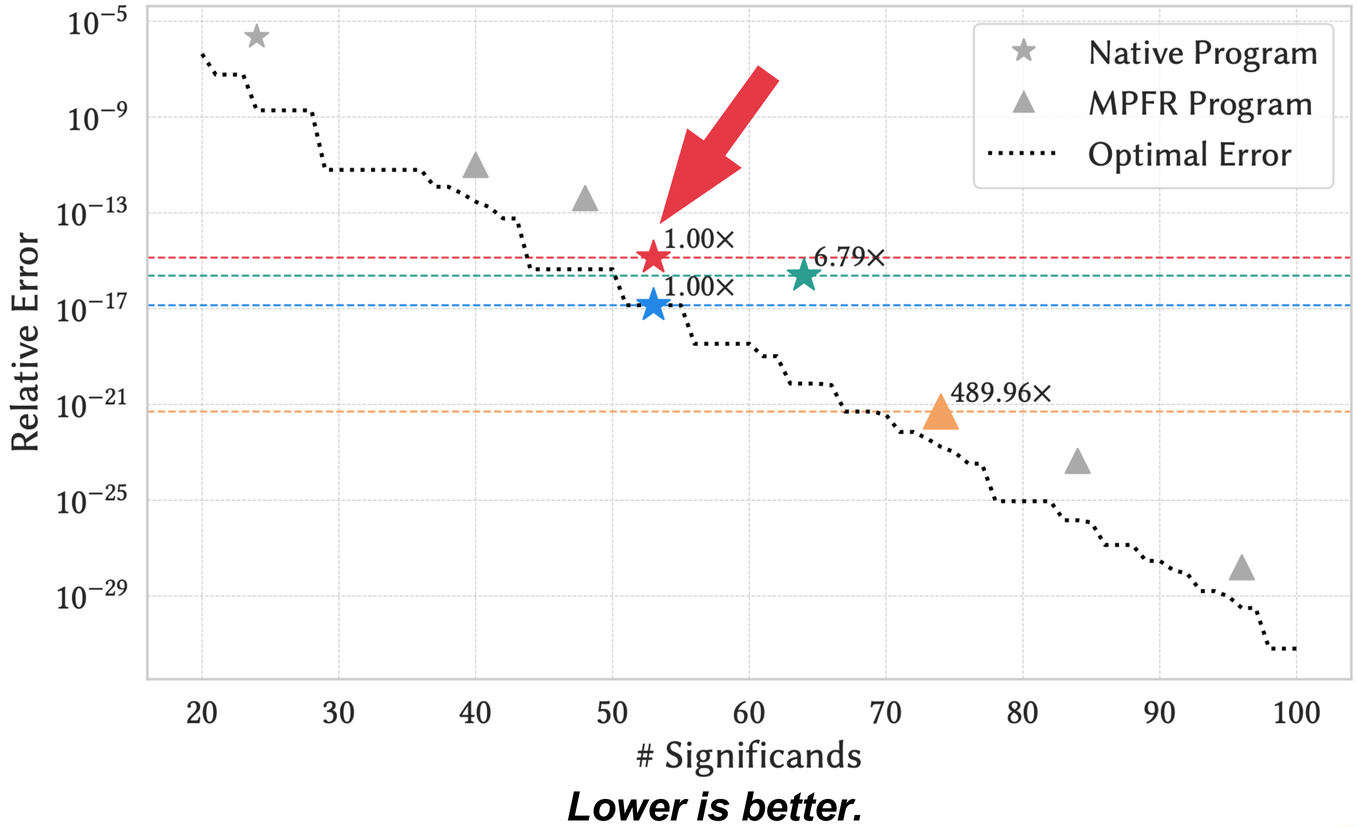
- Given the **original** program, Poseidon produces tradeoffs
- Trade accuracy for performance
  - **1.46×** with error of 6e-7
- Trade performance for accuracy
  - Error reduced by **27%** with 1.13× more compute time



Lower is more accurate; left is faster.

Evaluation: LULESH

- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off

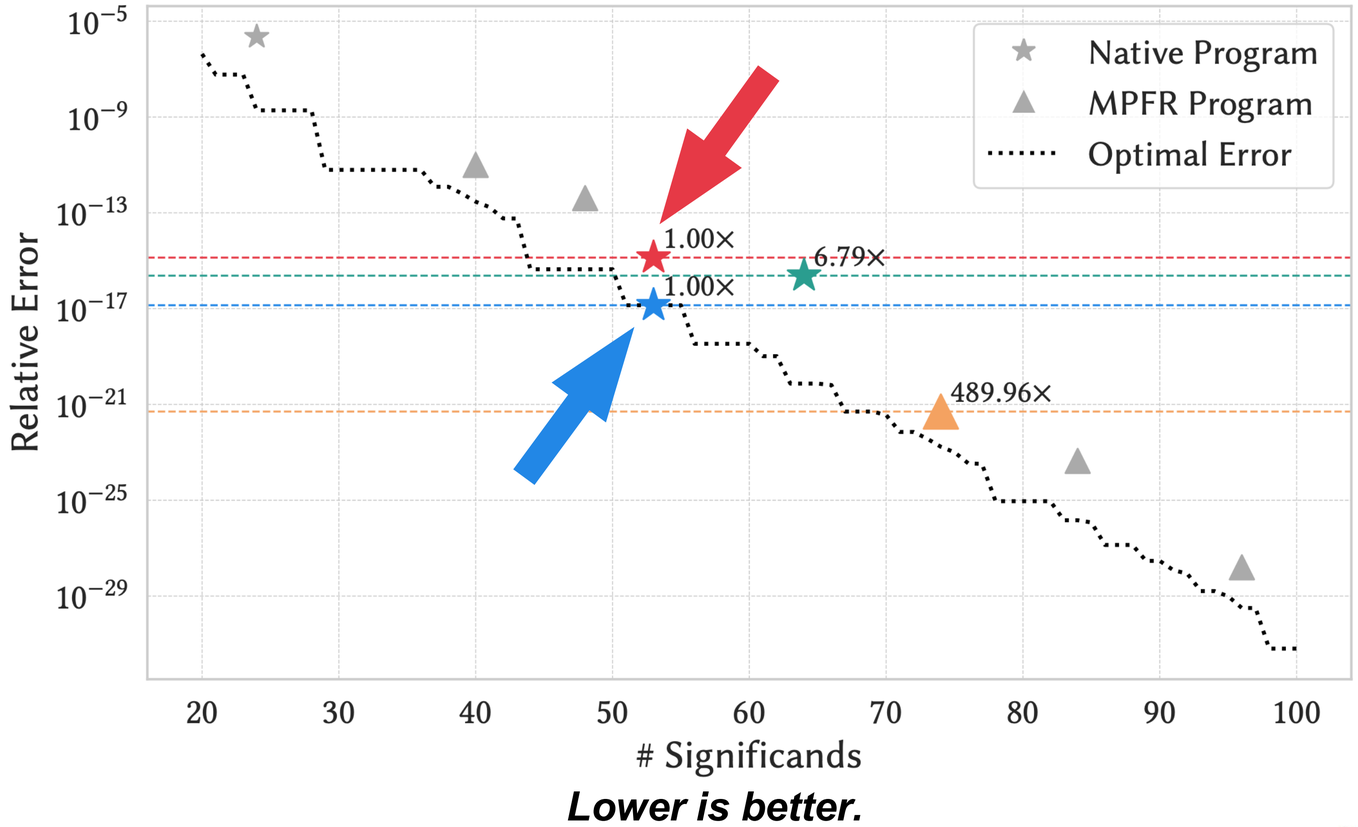


Lower is better.

* Ground truth: 64-bit rounding of MPFR-512 computation.

Evaluation: LULESH

- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off
- Poseidon (FP64): **Optimal* FP64 error**; little slowdown

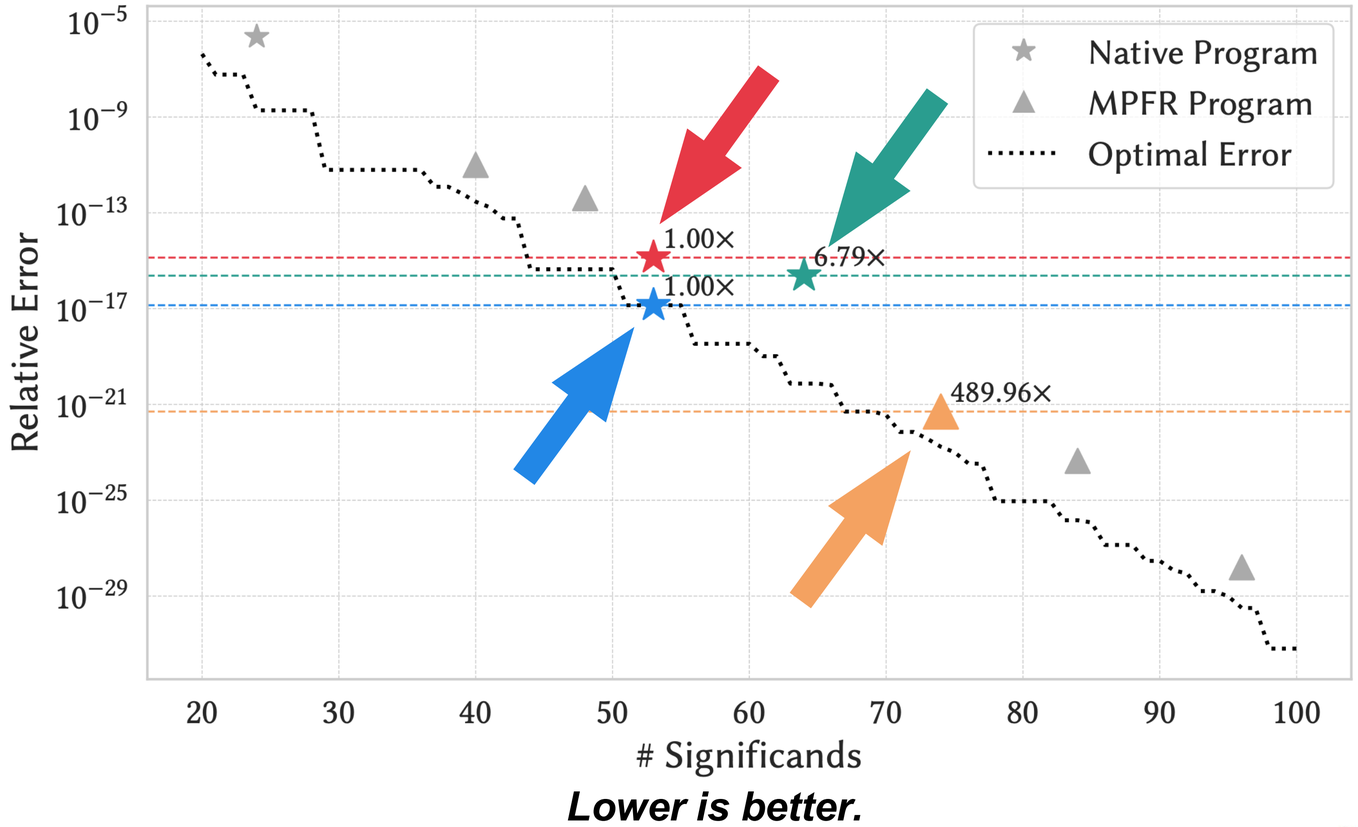


Lower is better.

* Ground truth: 64-bit rounding of MPFR-512 computation.

Evaluation: LULESH

- Large DOE proxy application that simulates Lagrangian hydrodynamics
- 5600+ LOC with over 200 loops
- Original (FP64): **12 ULPs*** off
- Poseidon (FP64): **Optimal* FP64 error**; little slowdown
- FP80: **6.79×** slower; still **2 ULPs*** off
- MPFR-74: **~500×** slower



Lower is better.

* Ground truth: 64-bit rounding of MPFR-512 computation.

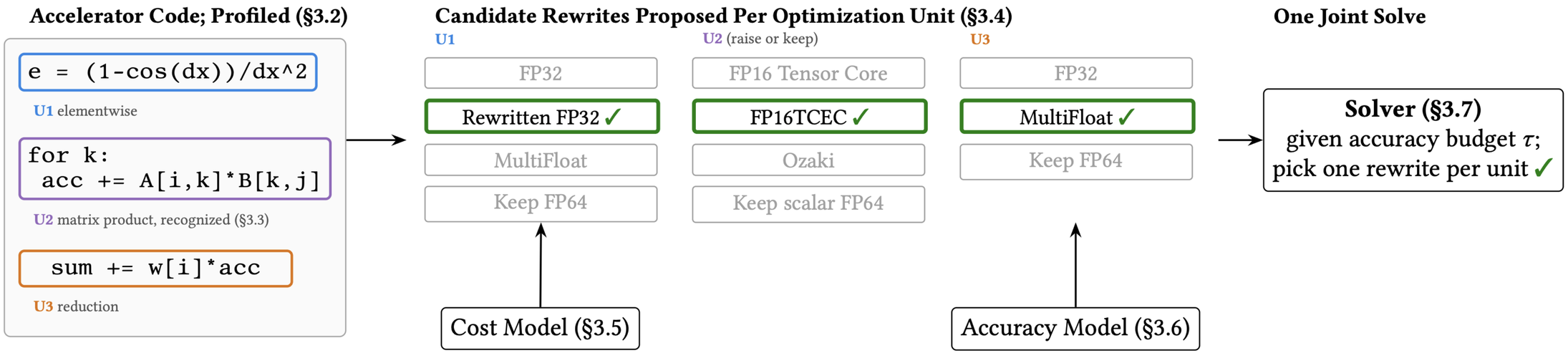
Ongoing Work: Poseidon for Accelerator Code Rewriting

- Fully leveraging arithmetic throughput of an accelerator is challenging
  - Most of FLOPS of an accelerator are in specialized hardware instructions (e.g., Tensor Core); raising requires expert analysis
  - A recent trend of deprioritizing high precision, e.g., GB202's FP64 runs at 1/64 rate of FP32
  - Different cost model: instead of using native FP64, it may be faster to emulate high precision arithmetic with lower-precision floats
  - Can we automatically choose the right rewrite strategy for practical accelerator code while still satisfying an application-level accuracy target?
- **Poseidon's profile-guided optimization approach is an answer**

Ongoing Work: Poseidon for Accelerator Code Rewriting

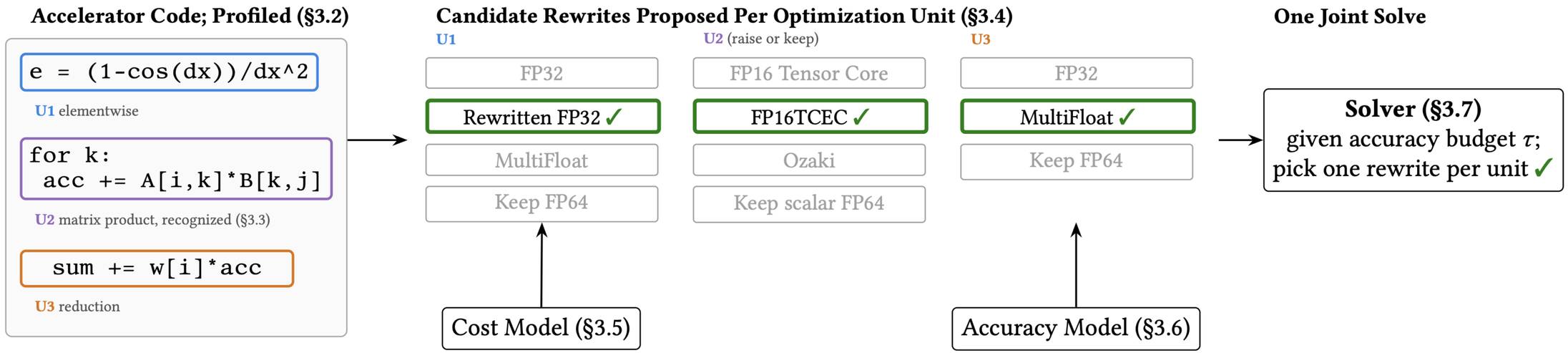
- Extended candidate generation phase with floating-point emulation techniques (e.g., MultiFloat, Ozaki scheme) and instruction-raising rewrites

- Given a user-specified accuracy target, Poseidon solves for the right rewrite strategy



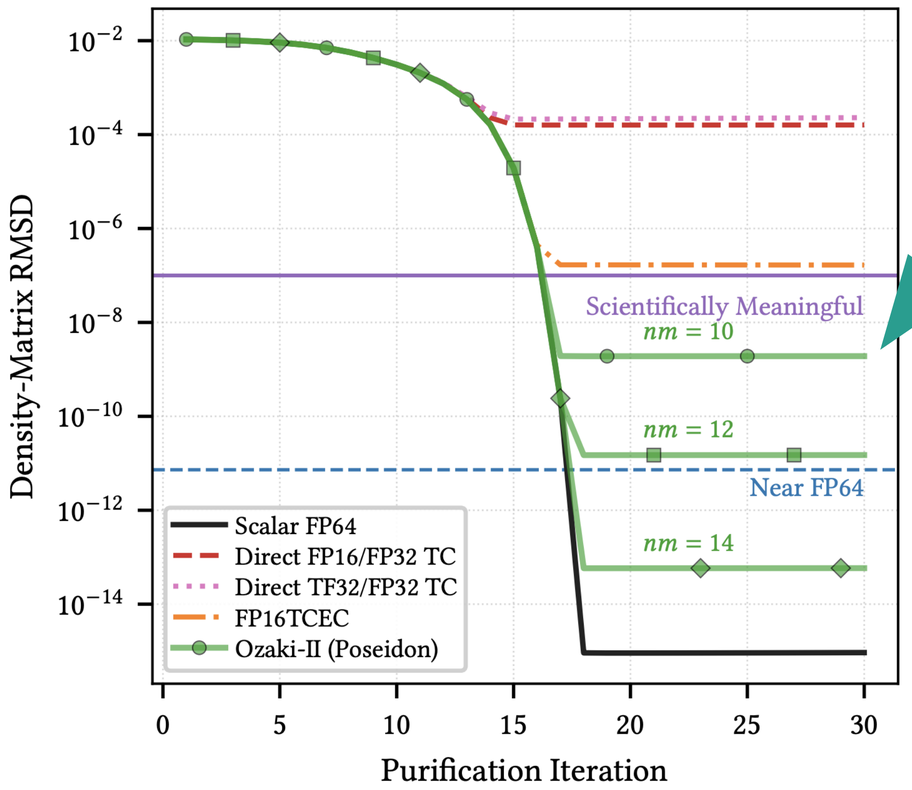
Ongoing Work: Poseidon for Accelerator Code Rewriting

- GENGA (astronomy): 4.4× faster, near-FP64 accuracy
- R-ChFSI (eigensolver): automatically raise an insensitive matrix product to TF32 Tensor Core that previously required expert analysis

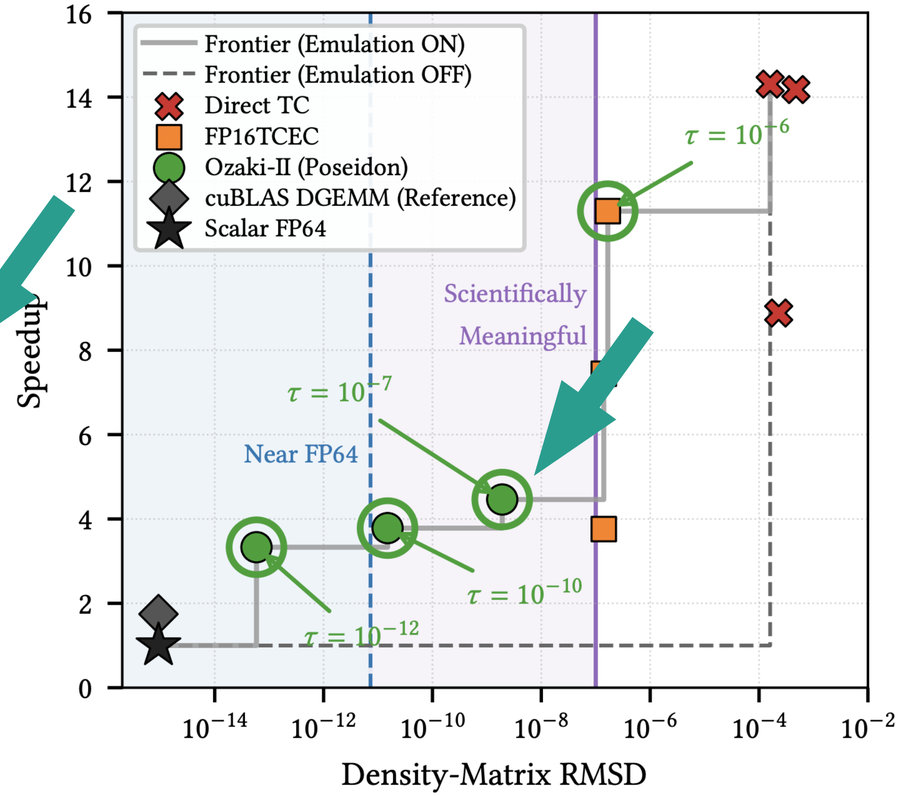


Ongoing Work: Poseidon for Accelerator Code Rewriting

- Ozp (chemistry): given a user-specified accuracy target required to produce scientifically meaningful results, Poseidon automatically raises scalar operations to Tensor Core and materializes Ozaki scheme II; 4.46× faster than scalar FP64 implementation



Lower is better.

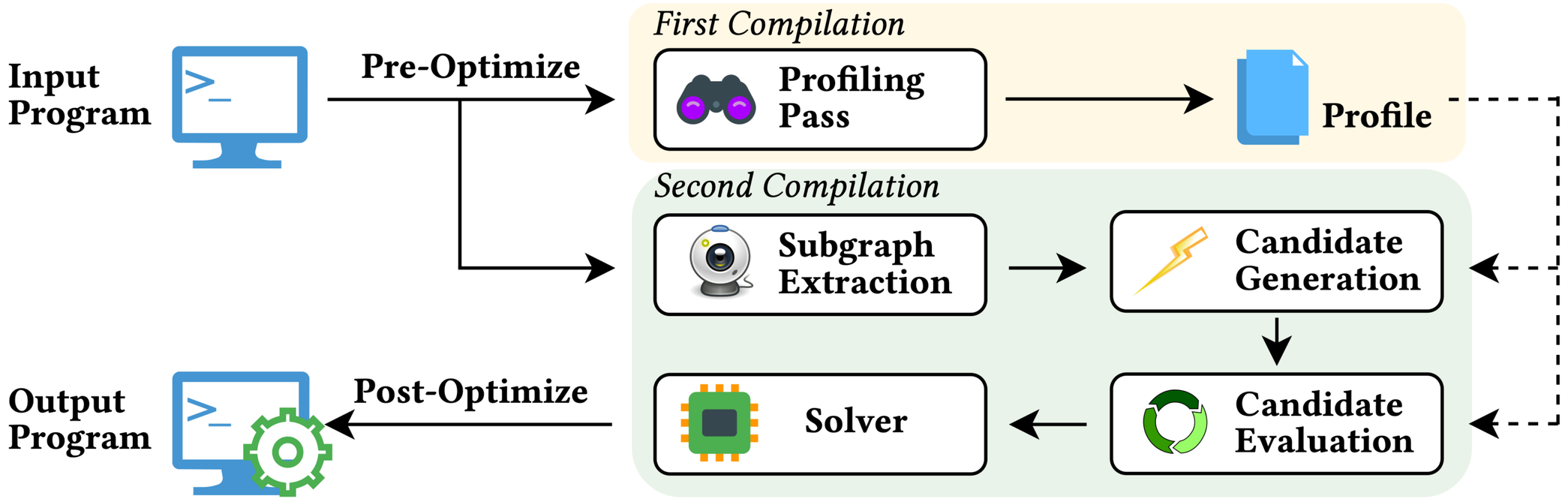


Upper-left is better.

Poseidon (Open Source on GitHub: **PRONTOLab/Poseidon**)

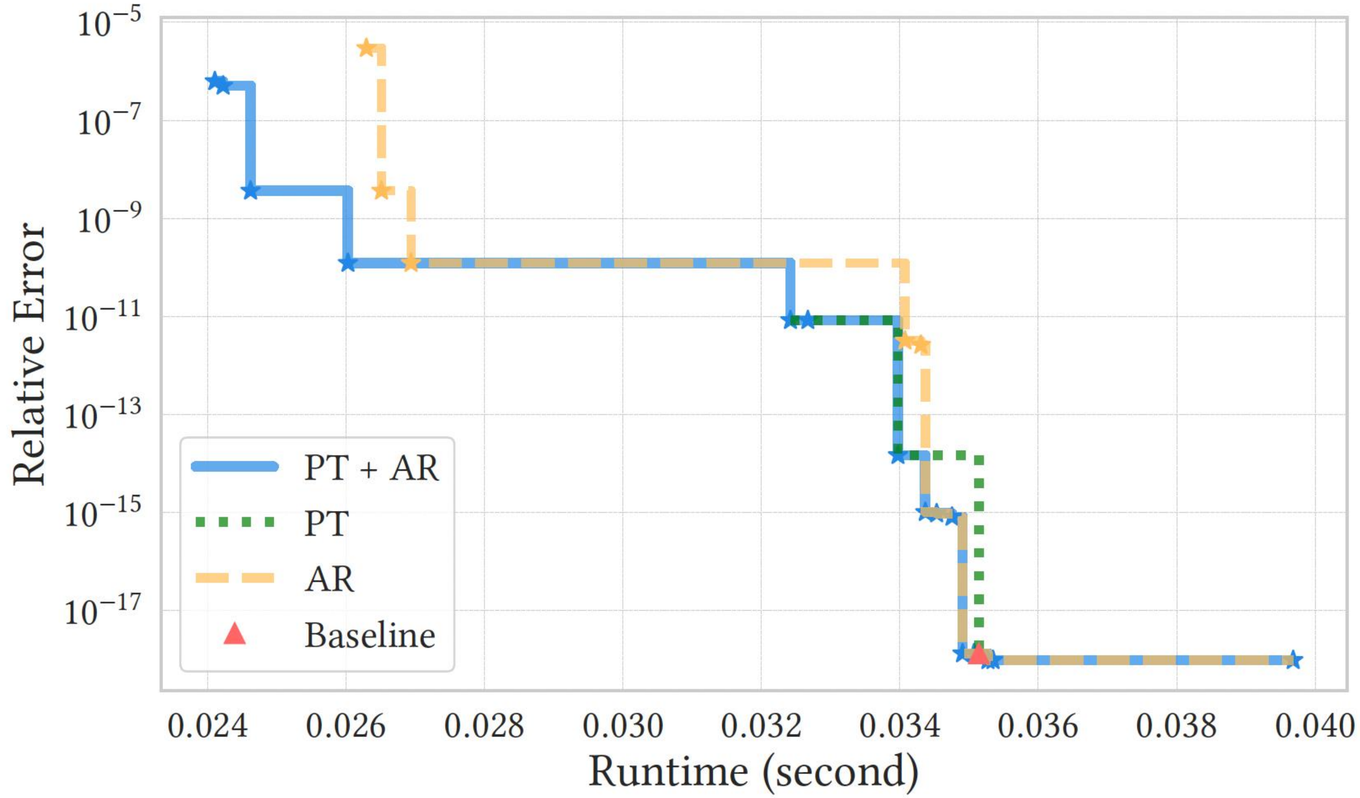
- **Profiling + Compiler Optimization** provides context, scope, and scale

- Automated numerical rewrites at a full-application scale within the compiler; recently, we extended Poseidon to enable fast and accurate accelerator code!



Evaluation: Quaternion Differentiator

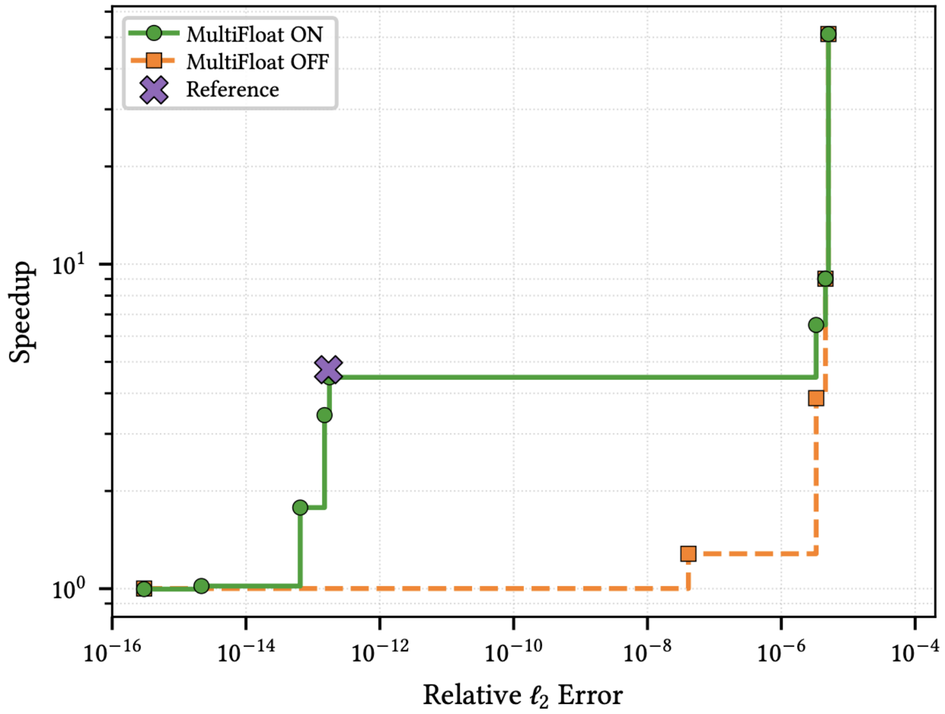
- PT + AR: Precision Tuning + Algebraic Rewrites
- PT: Precision Tuning only
- AR: Algebraic Rewrites only
- **PT + AR has the best frontier!**

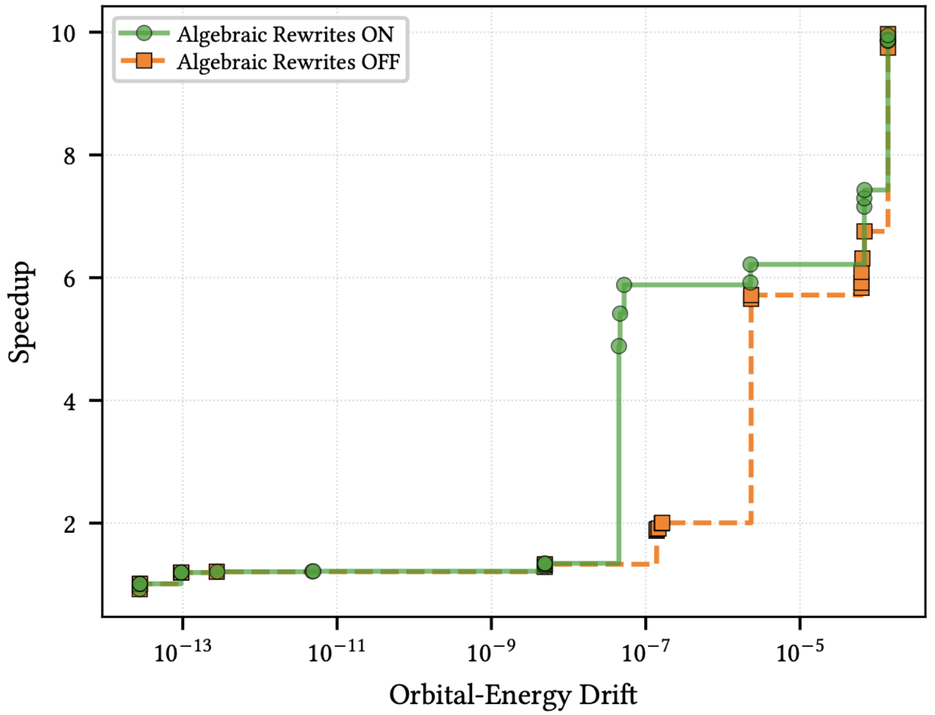


Lower-left is better.

Ongoing Work: Poseidon for Accelerator Code Rewriting

- GENGA (astronomy): 4.4× faster, near-FP64 accuracy





Ongoing Work: Poseidon for Accelerator Code Rewriting

- R-ChFSI (eigensolver): automatically raise a matrix product to TF32 Tensor Core that previously required expert analysis