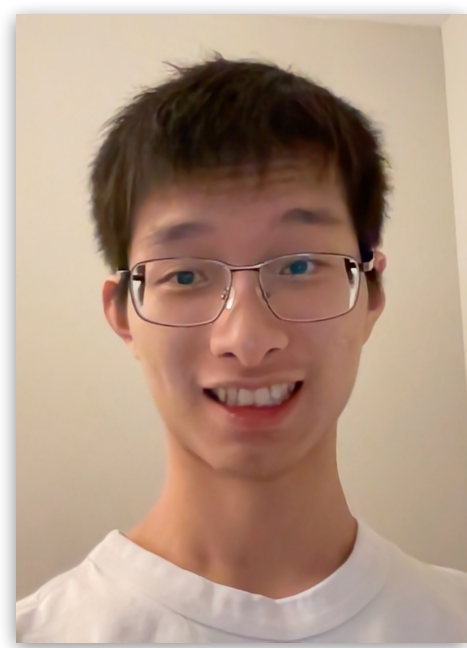


Thinking Fast and Correct: Automated Rewriting of Numerical Code through Compiler Augmentation



Siyuan Brant Qian¹



Vimarsh Sathia¹



Ivan R. Ivanov³



Jan Hückelheim²



Jens Domke³



Paul Hovland²



William S. Moses¹

¹ University of Illinois Urbana-Champaign, USA

² Argonne National Laboratory, USA

³ Institute of Science Tokyo and RIKEN CCS, Japan

FPTalks 2026

Aug 6, 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

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

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



Lower Activation:
1.6× speedup
error $\approx 1e-12$ (safe!)



Context: Where You Change Matters

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

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

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

Lower Activation:
1.6× speedup
error $\approx 1e-12$ (safe!)

Lower Reduction:
error $\approx 1e-4$ (danger!)



Context: Where You Change Matters

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

Profiling just 10 iterations:
Reduction is $10^4 \times$ more sensitive*!

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

Lower Activation:
1.6× speedup
error $\approx 1e-12$ (safe!)

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

Lower Reduction:
error $\approx 1e-4$ (danger!)

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



Context: *Structured Critical Value Hypothesis*

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

Profiling just 10 iterations:
Reduction is $10^4 \times$ more sensitive*!

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

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

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



Scope: Precision Isn't the Only Knob

FP32 (Original Order)

⚡ Fast

✗ Wrong

$(1e8 + 1) - 1e8 \rightarrow 1.00000000e8 - 1e8 \rightarrow 0$

+1 rounded away



Scope: Precision Isn't the Only Knob

FP32 (Original Order)

⚡ Fast

✗ Wrong

$(1e8 + 1) - 1e8 \rightarrow 1.00000000e8 - 1e8 \rightarrow 0$
+1 rounded away

FP64 (Same Order)

🐢 Costly

✓ Correct

$(1e8 + 1) - 1e8 \rightarrow 1.000000001e8 - 1e8 \rightarrow 1$

Use FP64
(Tradeoff!)



Scope: Precision Isn't the Only Knob

FP32 (Original Order)

⚡ Fast

✗ Wrong

$(1e8 + 1) - 1e8 \rightarrow 1.00000000e8 - 1e8 \rightarrow 0$

+1 rounded away

FP64 (Same Order)

🐢 Costly

✓ Correct

$(1e8 + 1) - 1e8 \rightarrow 1.000000001e8 - 1e8 \rightarrow 1$

Use FP64
(Tradeoff!)

FP32 (Reassociated)

⚡ Fast

✓ Correct

$(1e8 - 1e8) + 1 \rightarrow 0 + 1 \rightarrow 1$

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

```
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.



A production compiler makes math
explicit through optimizations
(**SimplifyCFG**, **mem2reg**, **inlining**, ...)



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.

```
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  
}
```



A production compiler makes math
explicit through optimizations
(**SimplifyCFG**, **mem2reg**, **inlining**, ...)



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.

```
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  
}
```

A production compiler makes math
explicit through optimizations
(**SimplifyCFG**, **mem2reg**, **inlining**, ...)

Aha, 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 numerical rewrites within a production compiler



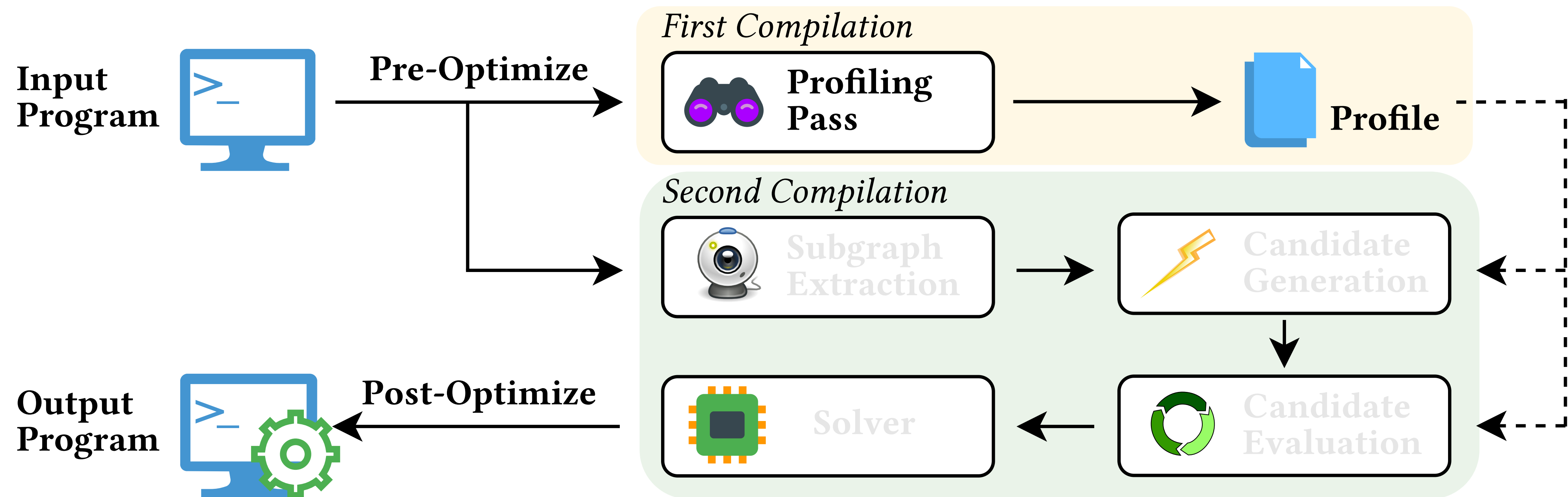
Our Answer — Poseidon

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



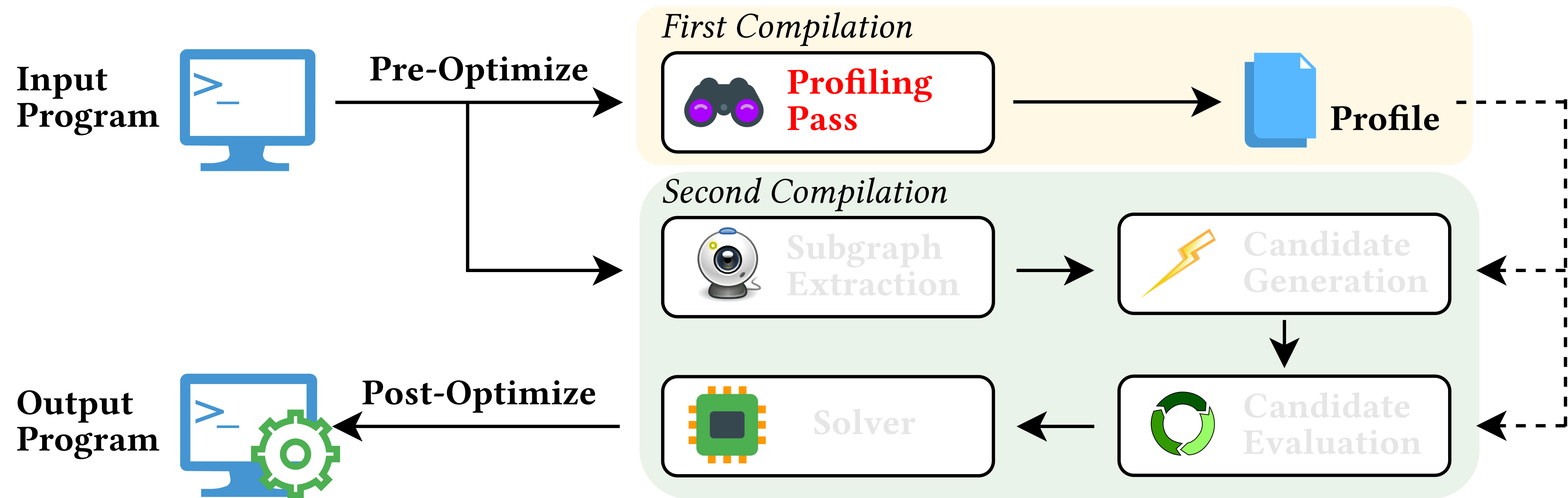
Our Answer — Poseidon

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



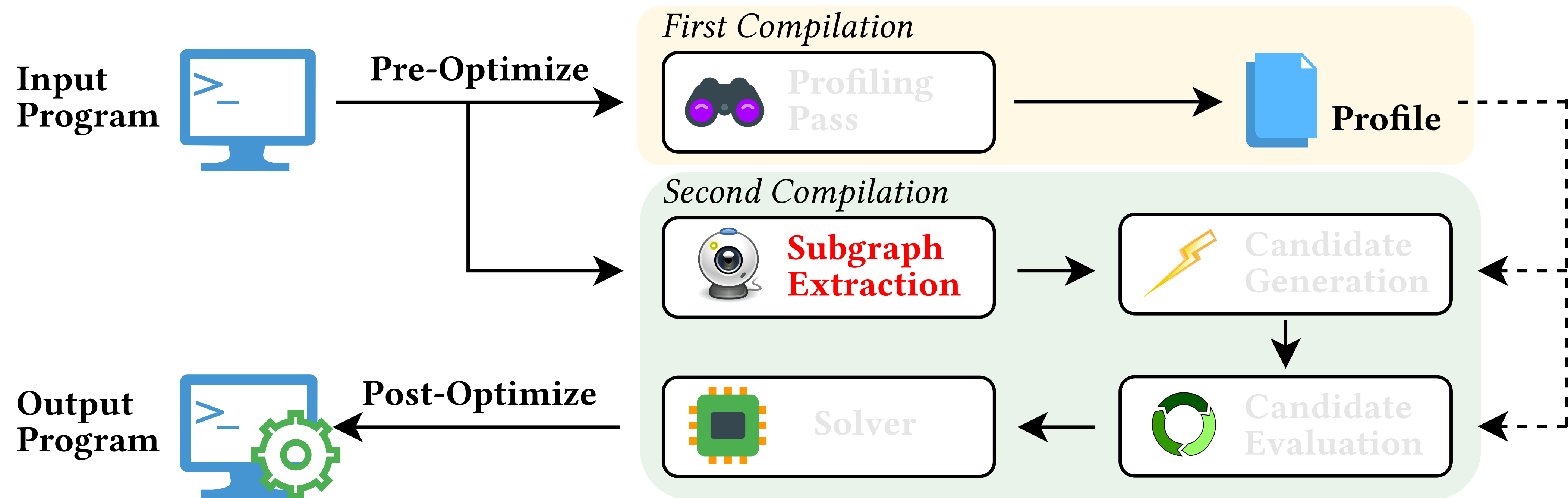
Profiling Pass

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



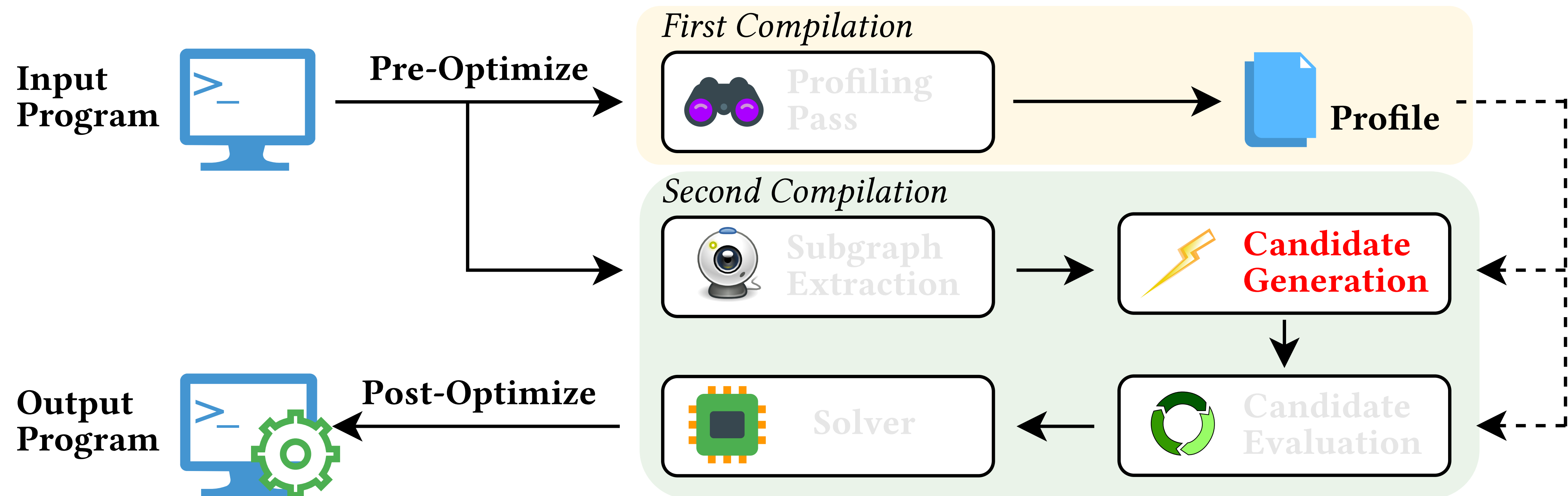
Subgraph Extraction

- Similarly, perform **pre-optimizations** to expose underlying math
- **Rewrite Regions:** floating-point def-use subgraph (linear-time flood fill)



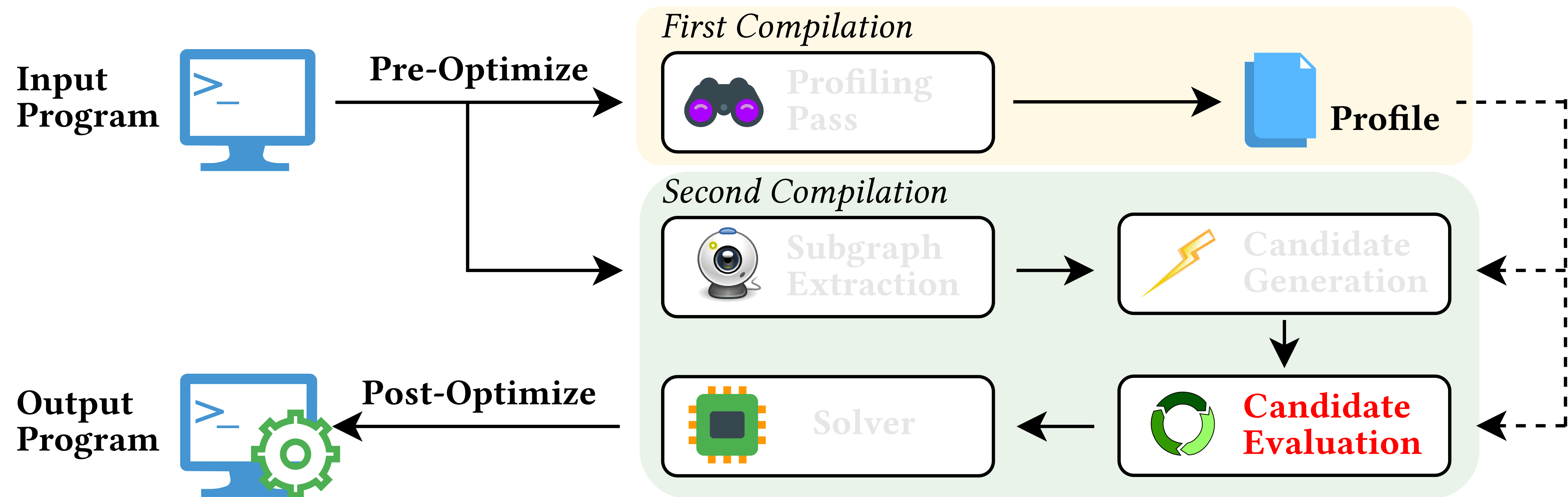
Candidate Generation

- Algebraic rewrites from external tools (e.g., Herbie)
- Profile-guided precision changes
- Fully extensible!



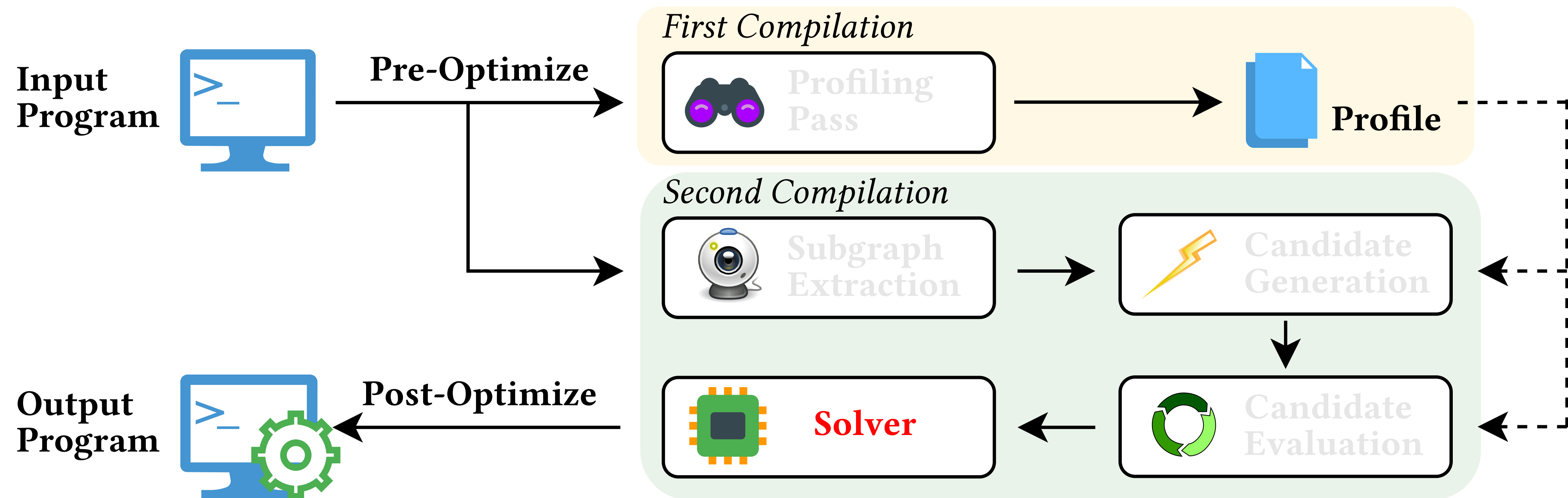
Candidate Evaluation

- Goal: **predict** candidate's performance/accuracy impact using the profile
- Cost: Sum over instructions of $(cost \times execution\ count)$
- Accuracy: Sum over instructions of $(local\ error \times global\ sensitivity)$



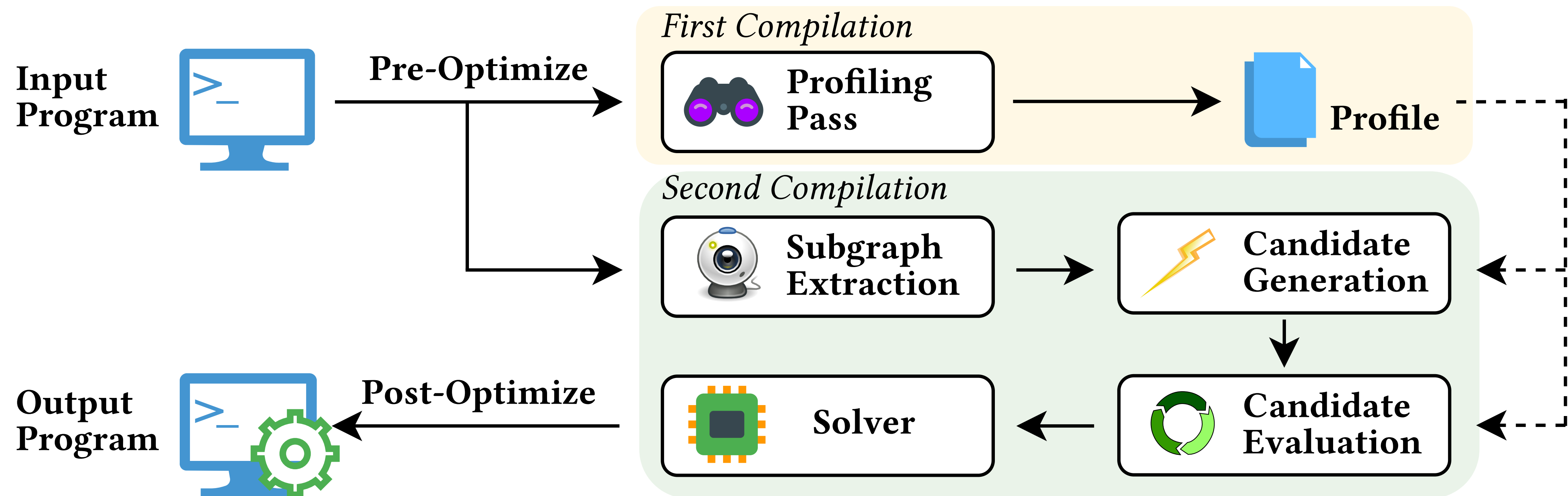
Solver (Global Selection of Rewrites)

- Objective: **minimize predicted global error** given a set of cost-error pairs
- Similar to *knapsack*: dynamic programming builds a **cost-error frontier**



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?



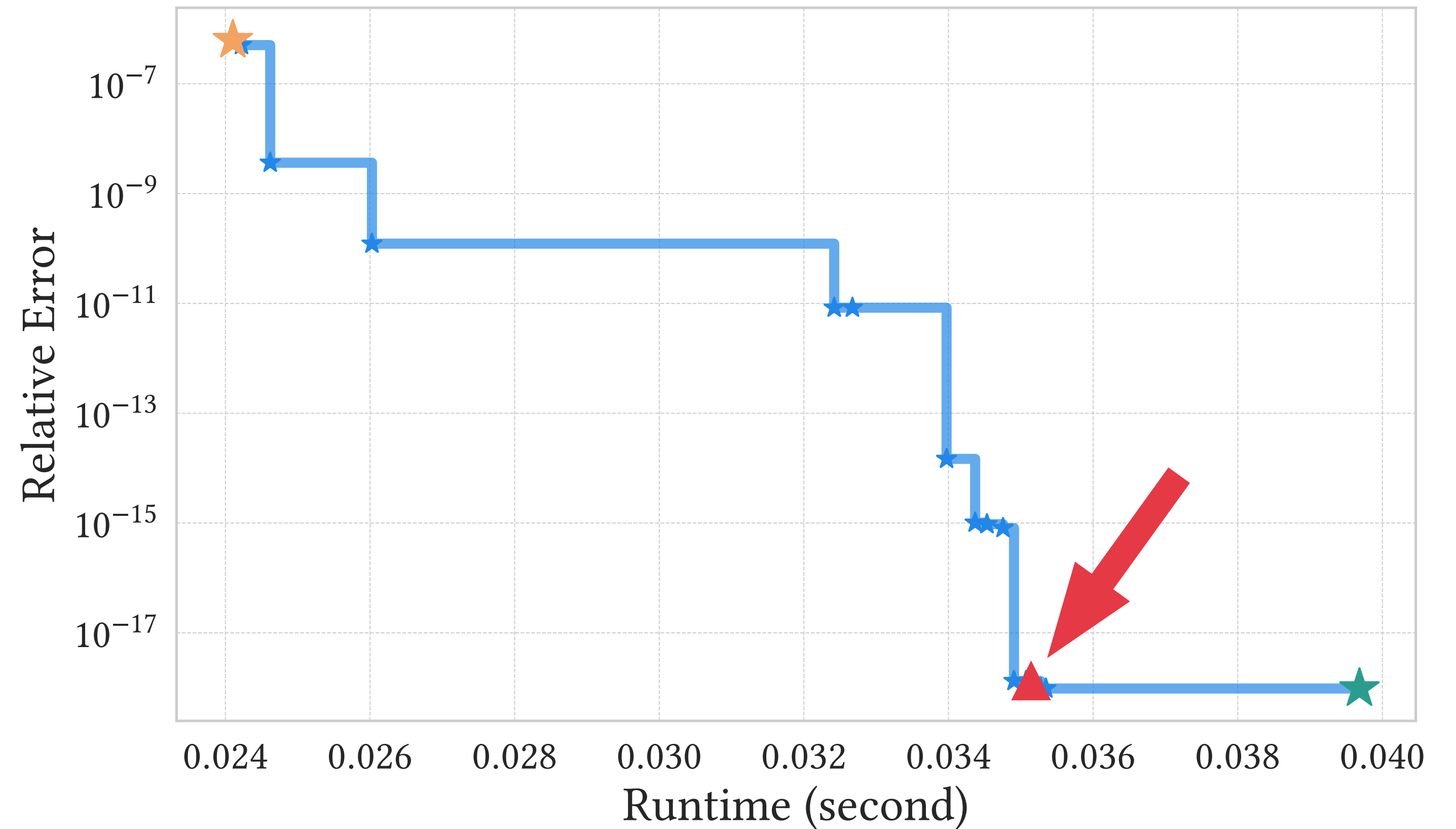
Evaluation

- Structured Critical Value Hypothesis: ***small surrogate profiles*** reveal numerical structure
- Coverage (FPBench): Accuracy improvements on **58%** of benchmarks;
Maximum speedup of **1.82×** (error < 1e-6)
- Quaternion Differentiator: **1.46×** speedup (error < 1e-6)
- LULESH: **Bitwise-identical to MPFR-512** in FP64; no substantial slowdown



Evaluation: Quaternion Differentiator

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

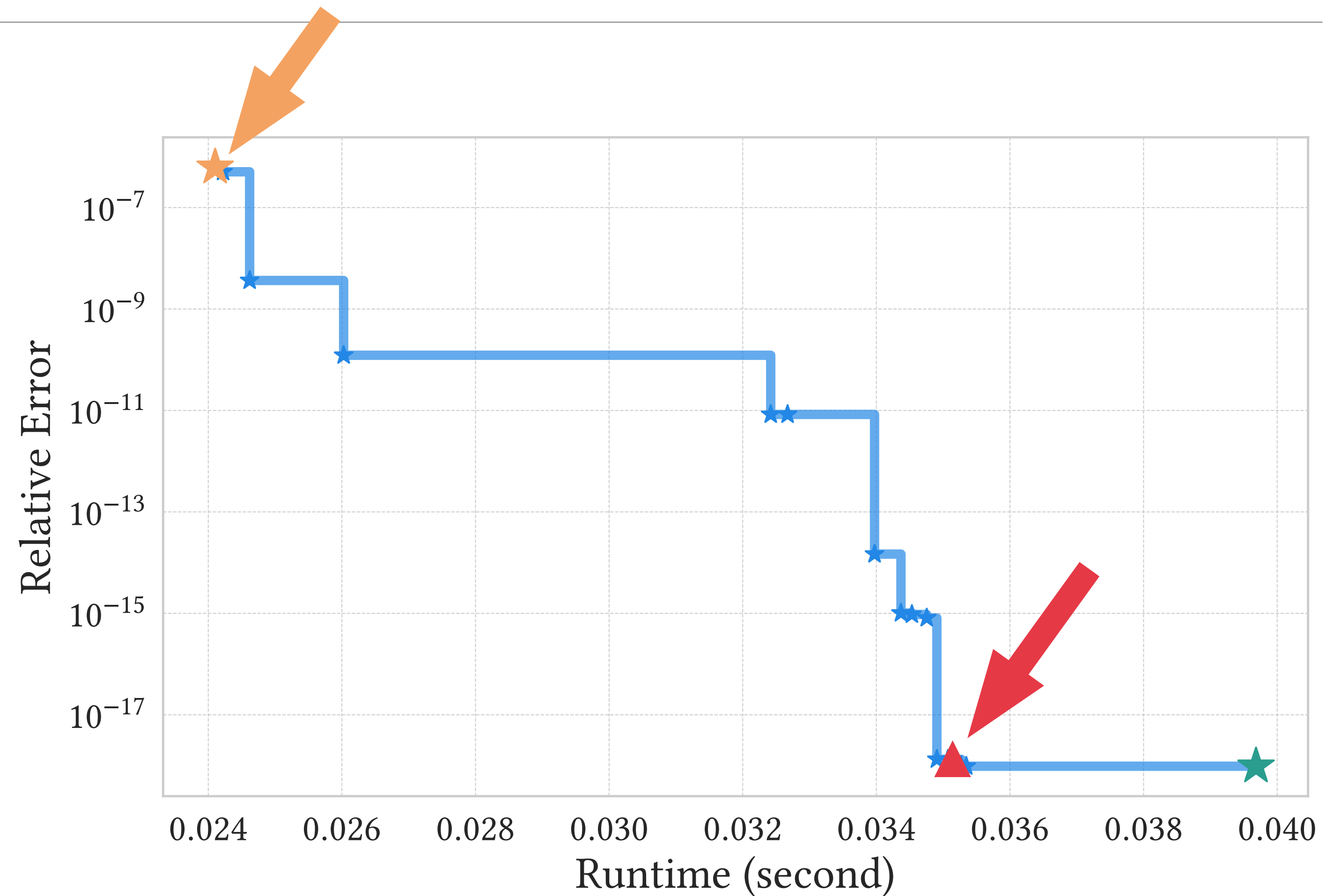


Lower is more accurate; left is faster



Evaluation: Quaternion Differentiator

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

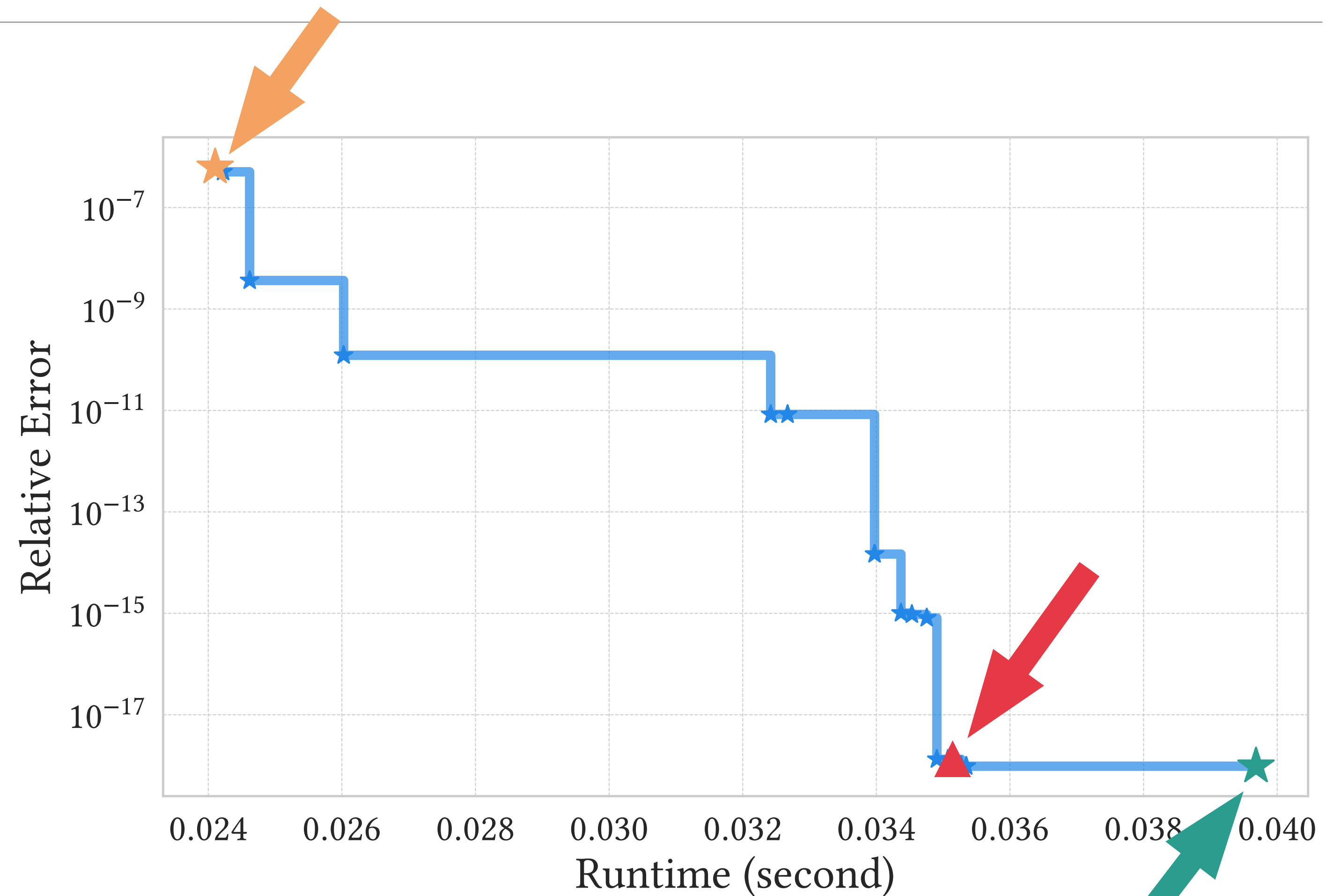


Lower is more accurate; left is faster



Evaluation: Quaternion Differentiator

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

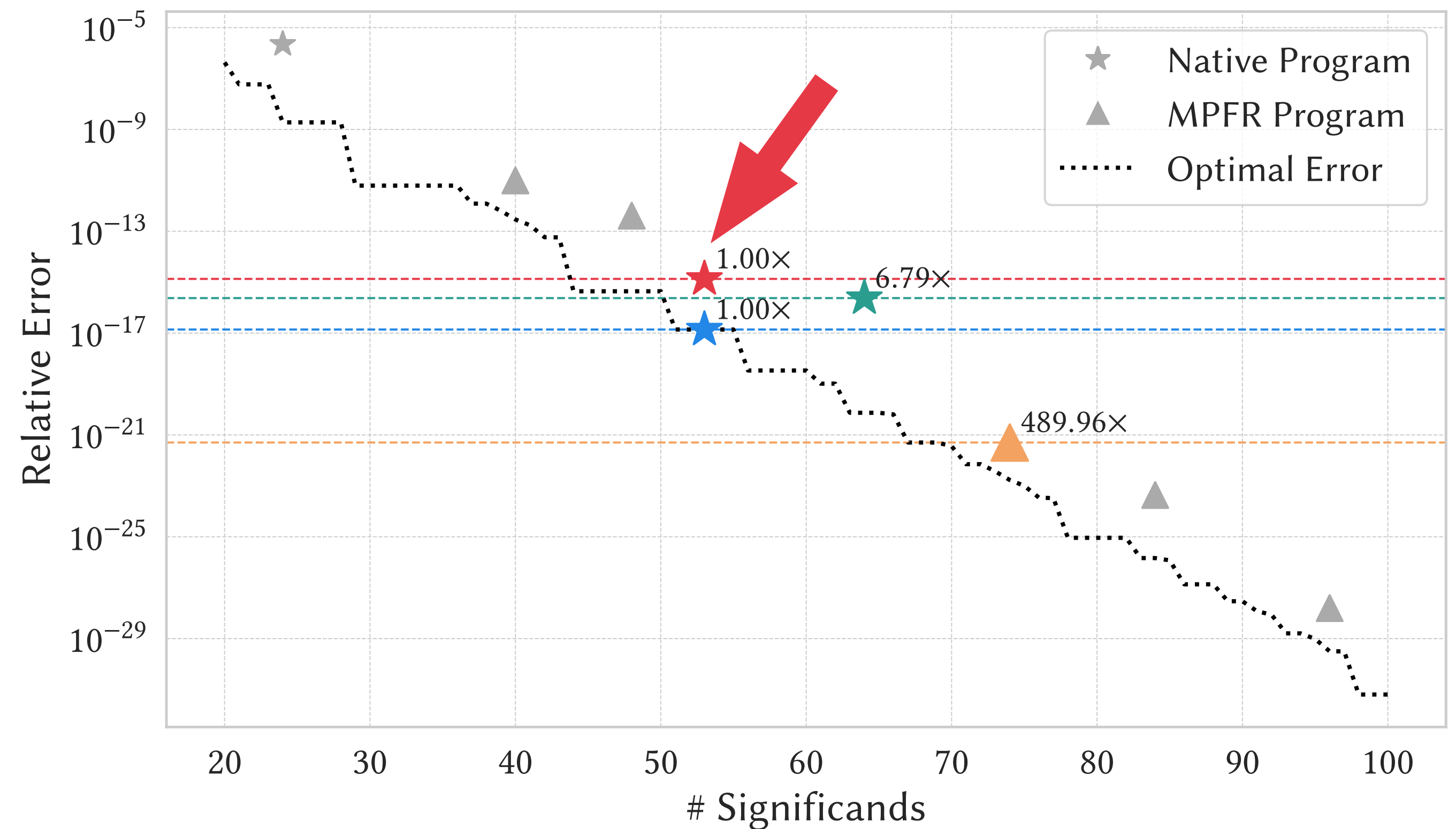


Lower is more accurate; left is faster



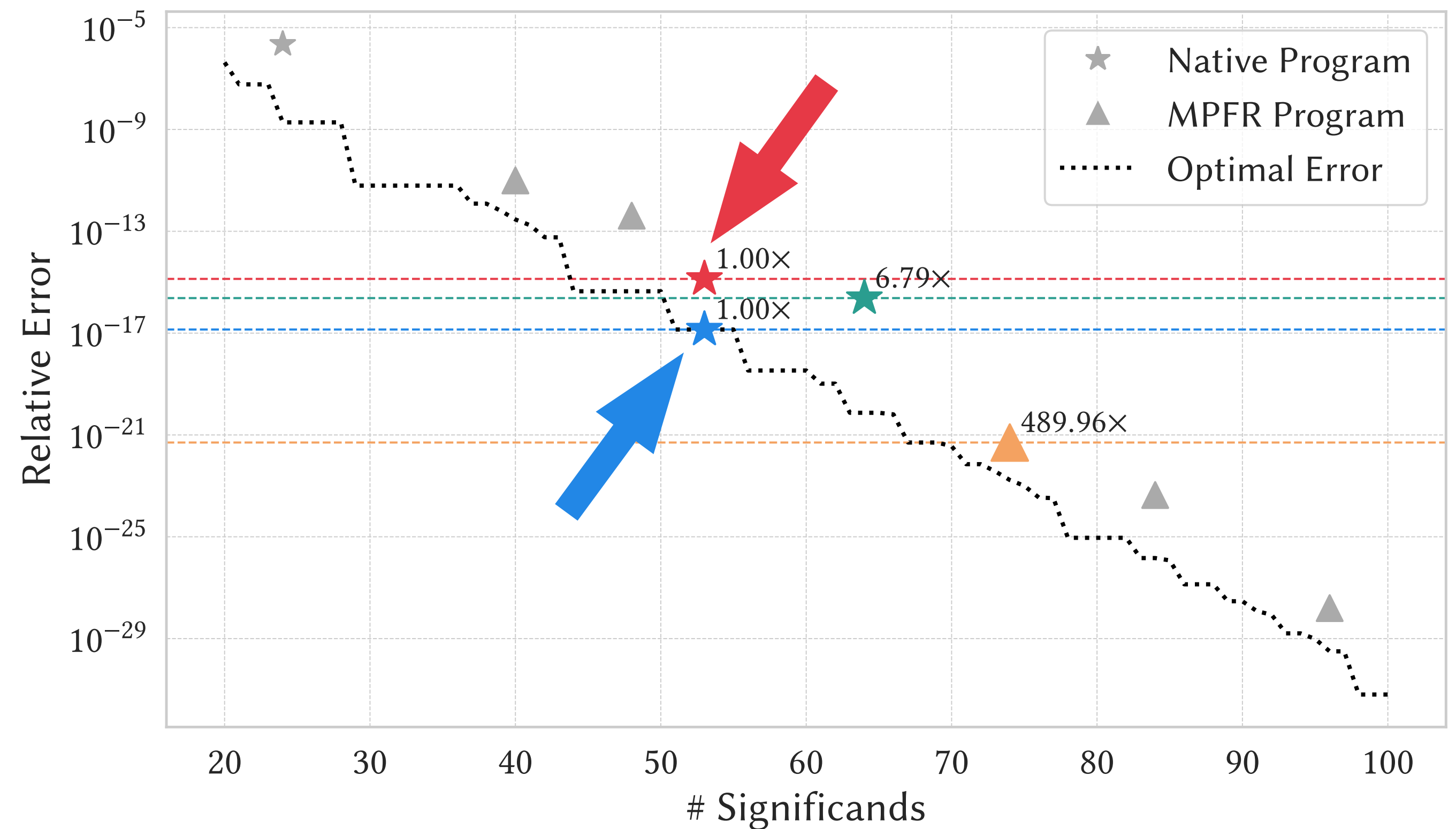
Evaluation: LULESH

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



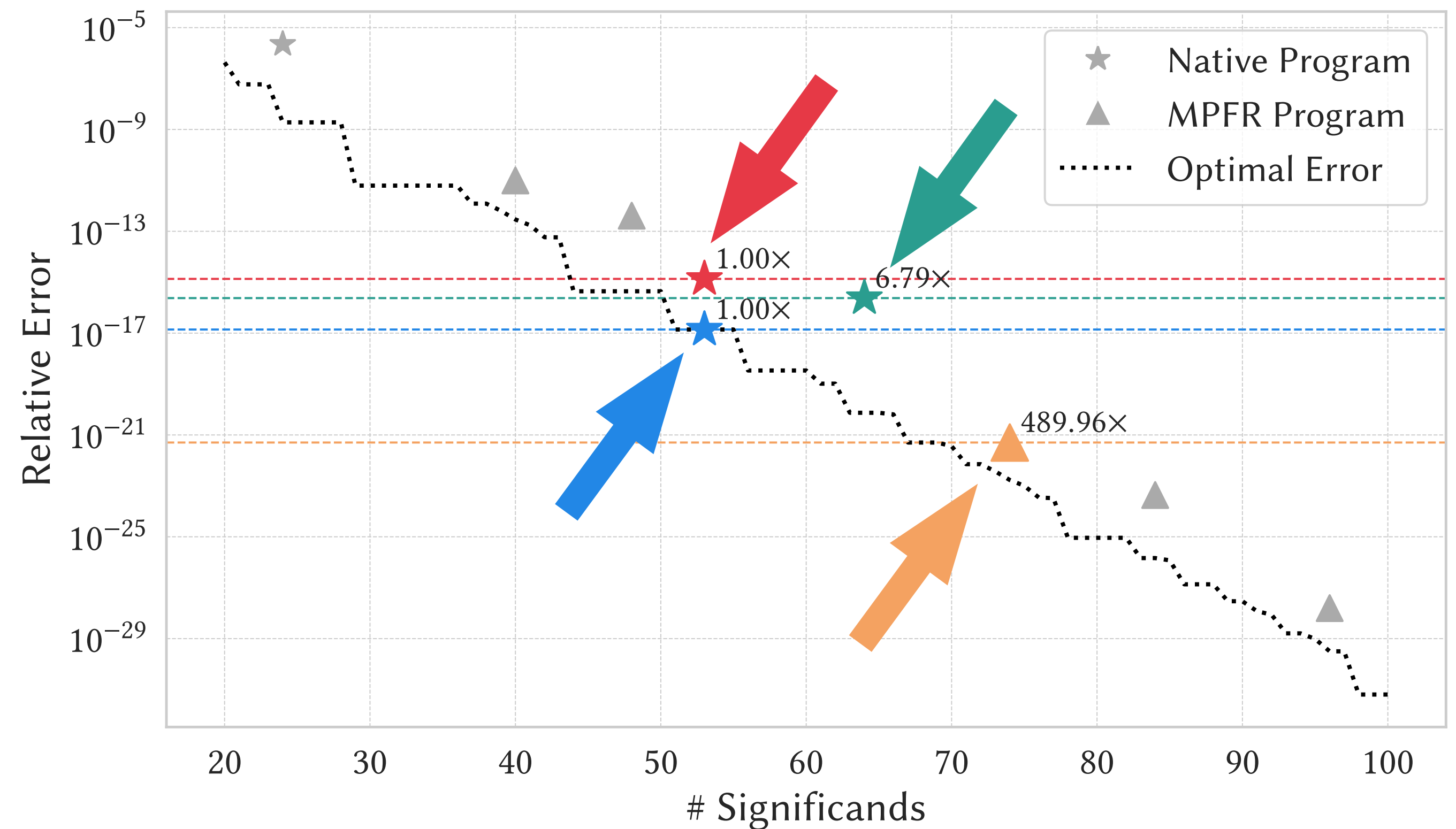
Evaluation: LULESH

- 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



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: **6.79x** slower; still **2 ULPs*** off
- MPFR-74: **~500x** slower



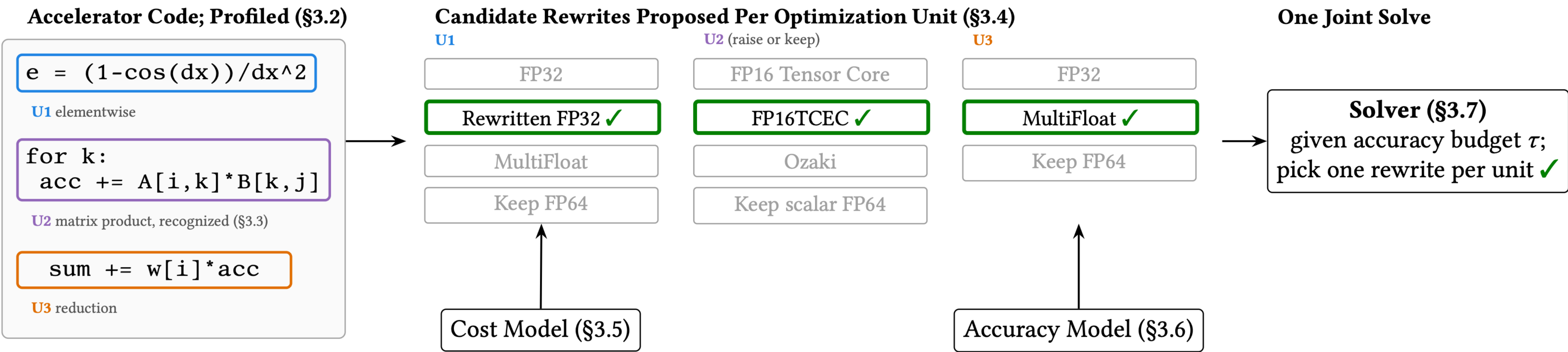
Ongoing Work: Poseidon for Accelerator Code Rewriting

- Fully leveraging arithmetic throughput of an accelerator is challenging
 - Most FLOPS in an accelerator are in specialized hardware instructions (e.g., Tensor Core)
 - High precision instructions are de-prioritized, 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**



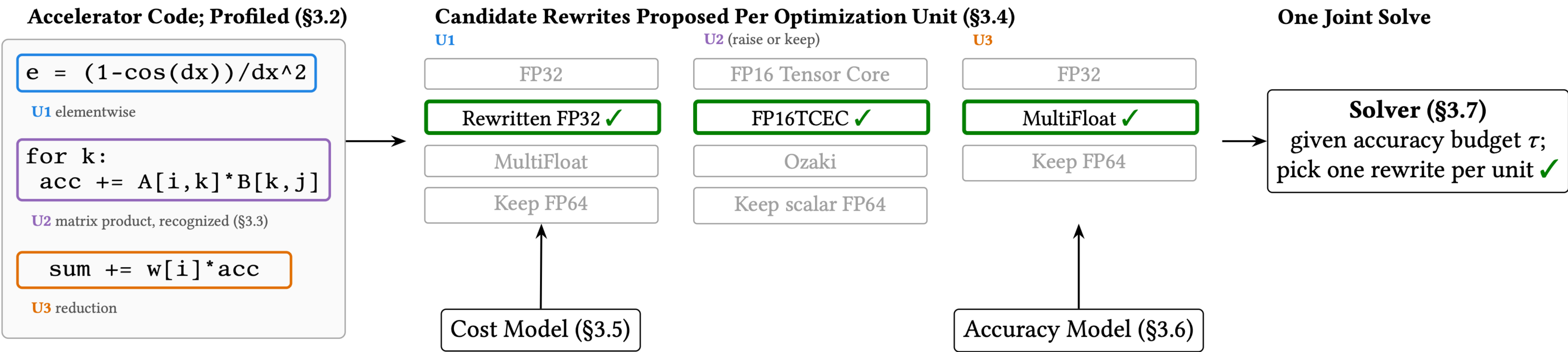
Ongoing Work: Poseidon for Accelerator Code Rewriting

- 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



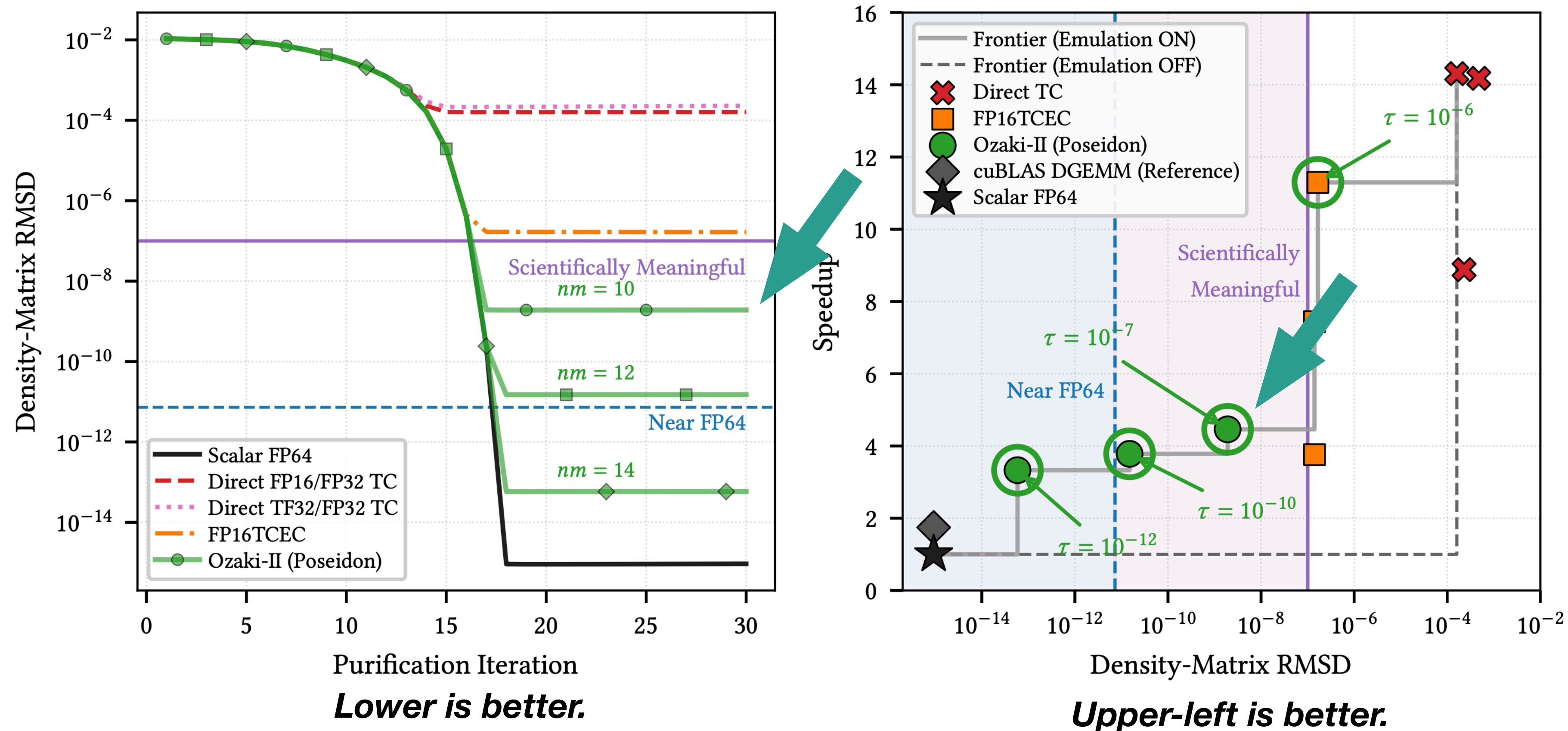
Ongoing Work: Poseidon for Accelerator Code Rewriting

- GENGA (astronomy): 4.4× faster, retains 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

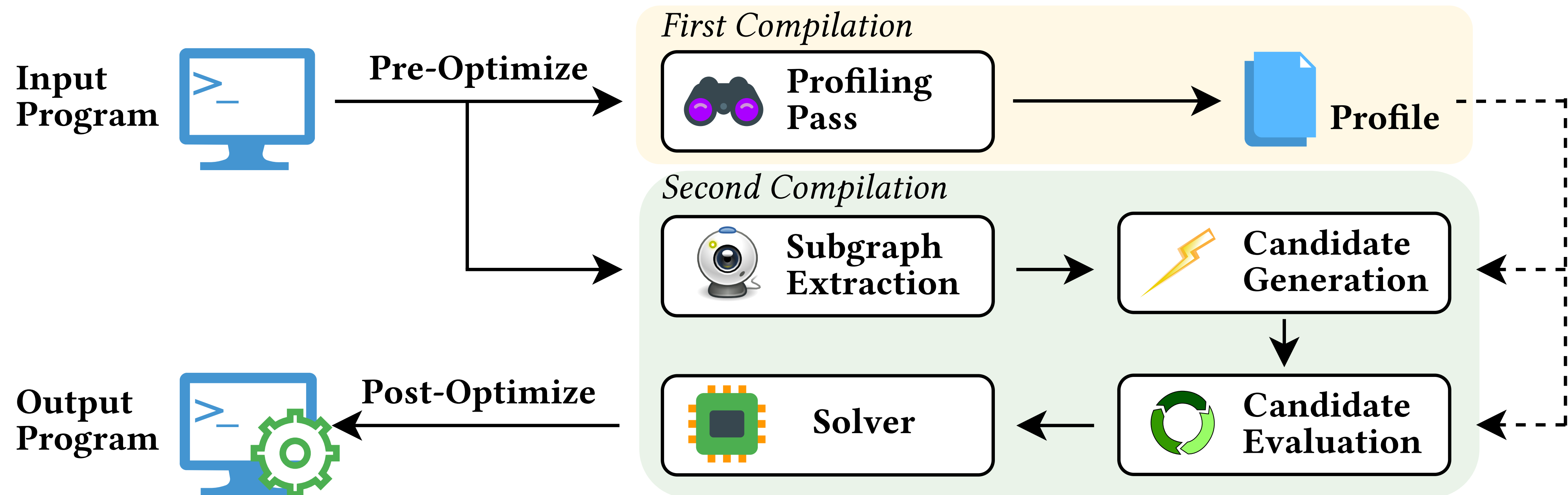
- Ozp (chemistry): given a user-specified accuracy target required to produce scientifically meaningful results, Poseidon automatically raises scalar operations to Tensor Core and materializes Ozaki scheme II; 4.46× faster than scalar FP64 implementation





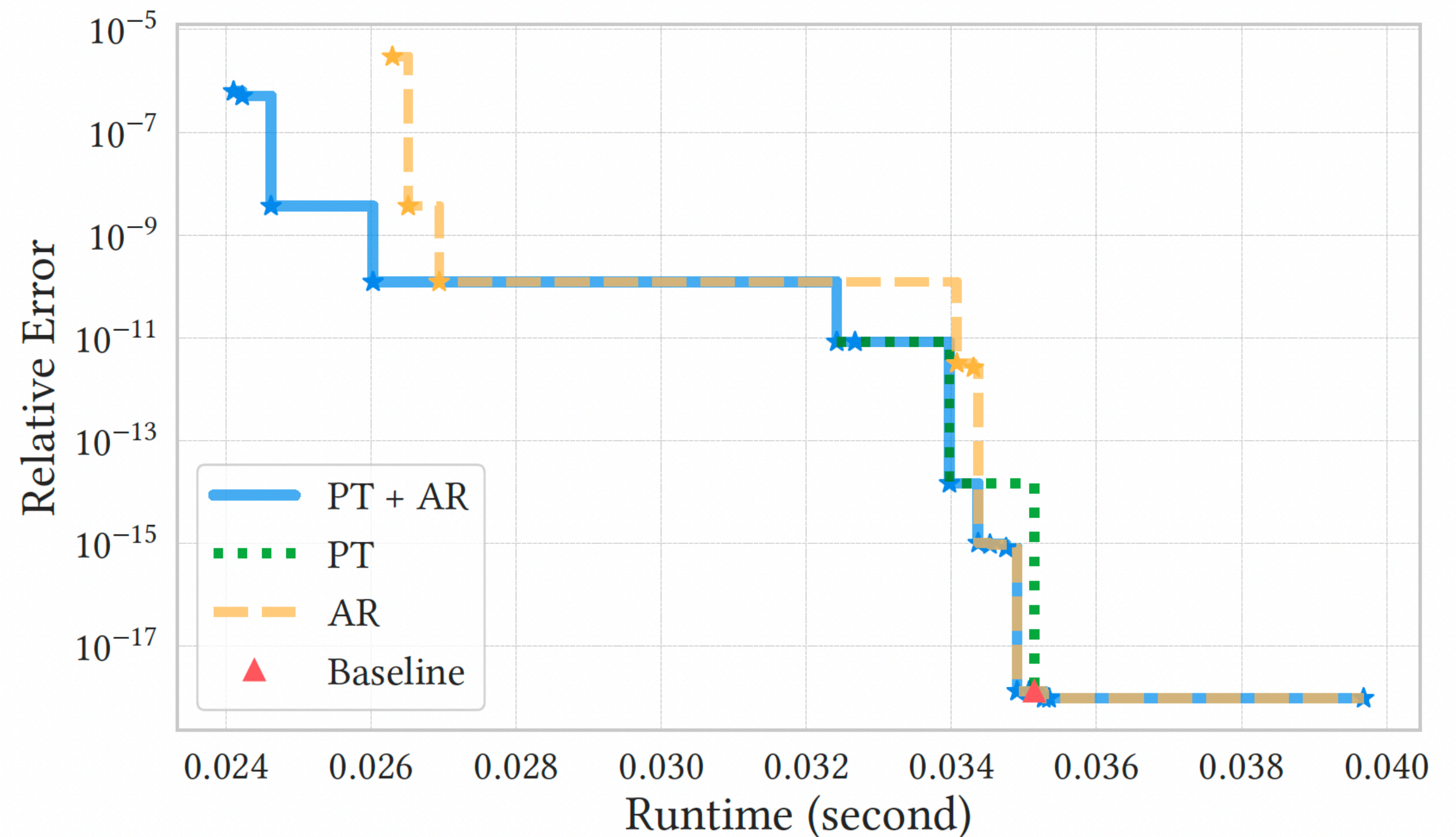
Poseidon (Open Source on GitHub: [PRONTOLab/Poseidon](https://github.com/PRONTOLab/Poseidon))

- The first framework that automates numerical rewrites within a production compiler
- **Profiling + Compiler Optimization** provides context, scope, and scale
- Outsized performance benefit without sacrificing the accuracy, and *vice versa*



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



Evaluation: Artificial Bound Increases

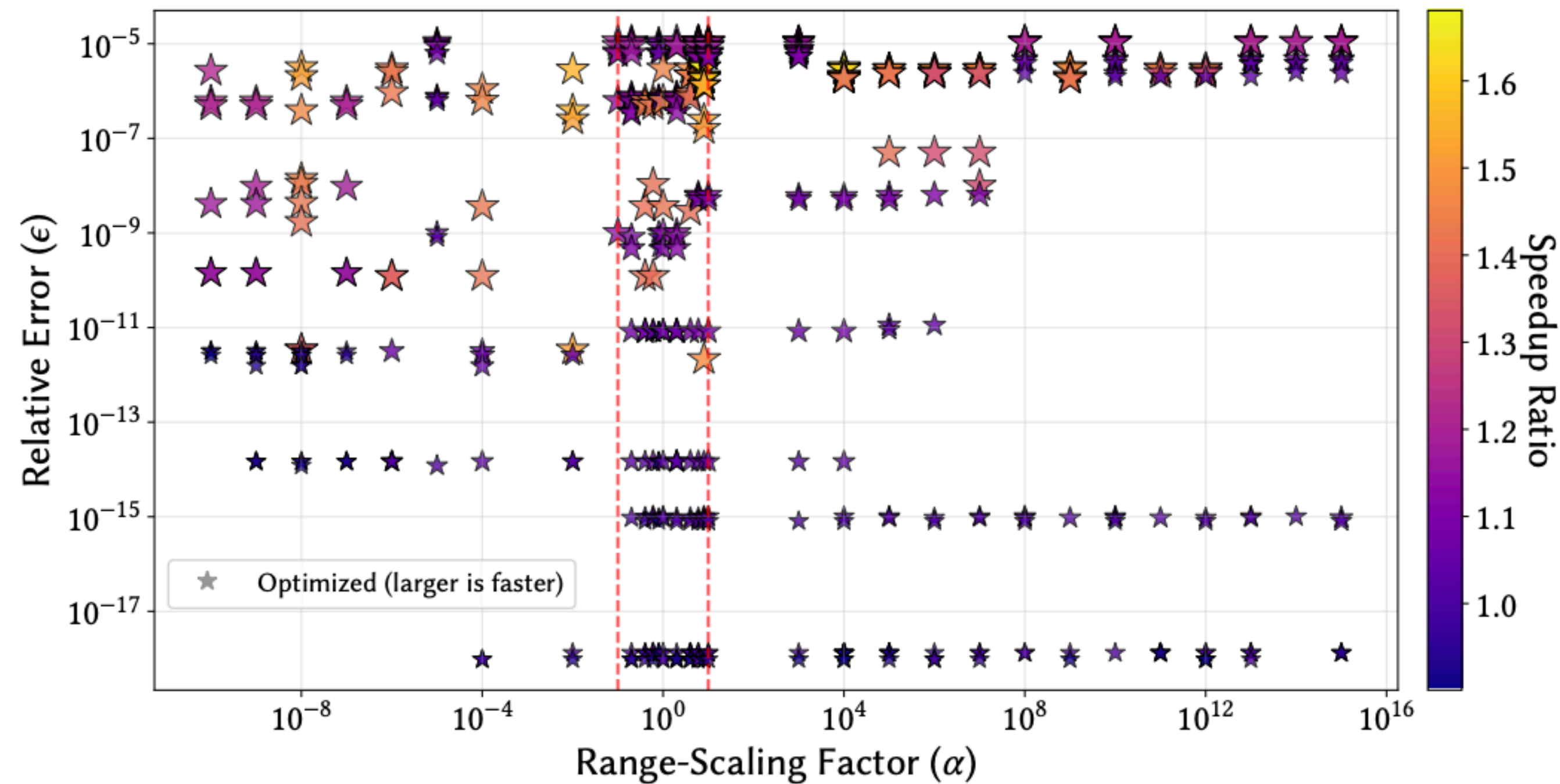


Fig. 11: Optimized dquat programs under varying range-scaling factors. Stars represent Pareto-optimal programs. Vertically aligned stars share the same factor. Lighter hues and larger stars indicate better runtime performance. Red dashed lines represent $\alpha = 0.1$ and $\alpha = 10$.

