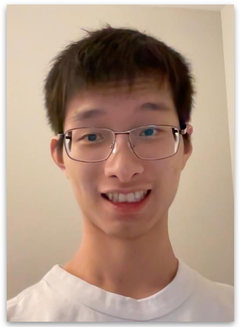
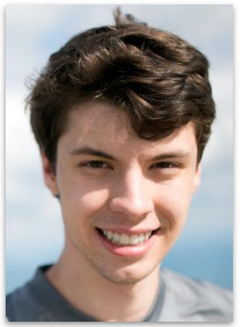
Impulse: Momentously Fast, General, and Portable Probabilistic Programming via Compiler Augmentation





Siyuan Brant Qian[1] Vimarsh Sathia[1] Jesse Michel[2] William S. Moses[1]

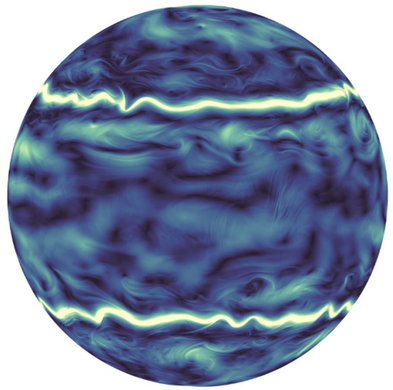
siyuanq4@illinois.edu

[1] University of Illinois Urbana-Champaign, USA
[2] Massachusetts Institute of Technology, USA

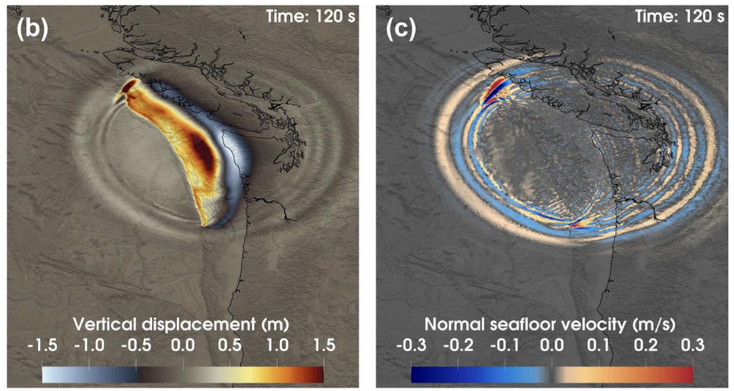
LLVM Developers' Meeting, Santa Clara, CA
October 27, 2026

Bayesian Inference and Probabilistic Programming

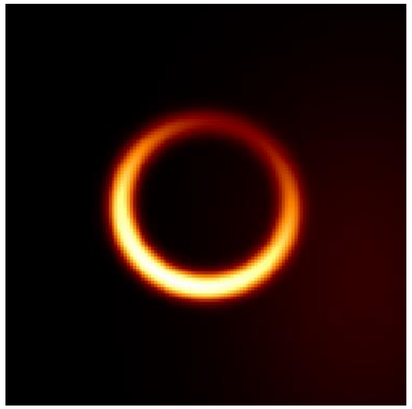
- **Bayesian inference** infers unknown values from observed data
- **Probabilistic programming** frameworks automates Bayesian inference by taking a **statistical model specification** and generate code that infers the unknowns



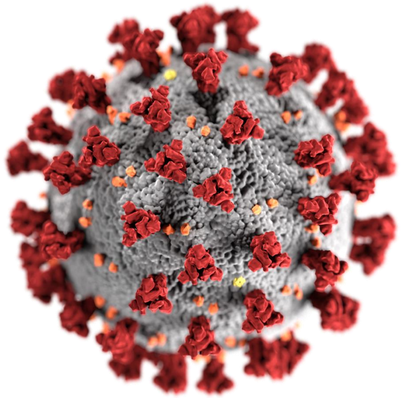
Earth Systems Modeling
Credit: Moses et al., JAMES 2026



Tsunami Early Warning
Credit: Henneking et al., SC '25



Black Hole Imaging
Credit: Comrade.jl, GeometricModeling tutorial



COVID-19 Vaccine Trials
Credit: CDC / Alissa Eckert & Dan Higgins

Code you want to **write** ≠ code you want to **run**.

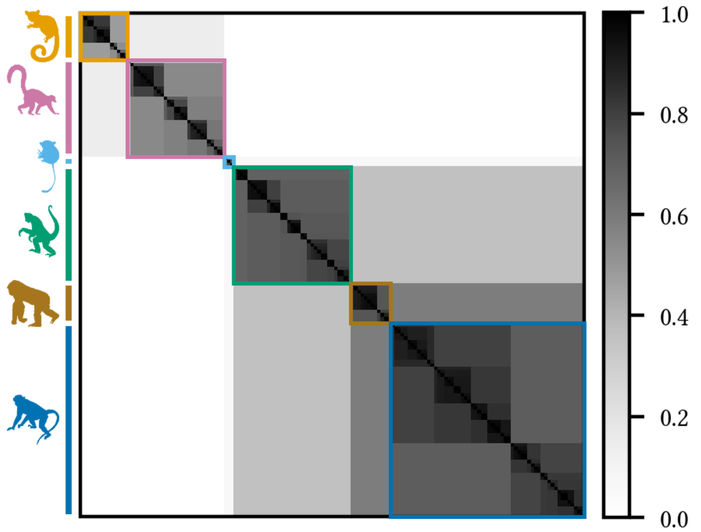
Give the **compiler** first-class probabilistic primitives to automatically enable **fast Bayesian inference**!

Example: model specifications often incur redundant work

- A common source is the **direct transcription** of a textbook formulation

Do primates in larger groups have larger brains?

```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```



`R`: fixed 151×151 biological data

Source: McElreath, Statistical Rethinking, 2nd edition, Section 14.5, https://doi.org/10.1201/9780429029608

The framework compiles the model into a hot loop

- Every loop iteration evaluates the **log-density** of the model and its **gradient**

Code you want to write

```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```

```
for sample in range(num_samples):
  for step in range(num_steps):
    logp_fn = logdensity(model)
    logp, grad = value_and_grad(run)(
        logp_fn, inputs, unknowns
    )
    unknowns = update(logp, unknowns, grad)
```

A hidden $\Theta(n^3)$ Cholesky dominates every iteration

- **MvNormal** internally computes $\text{cholesky}(\sigma^2 \cdot R)$

Code you want to write

```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```

```
for sample in range(num_samples):
  for step in range(num_steps):
    logp_fn = logdensity(model)
    logp, grad = value_and_grad(run)(
        logp_fn, inputs, unknowns
    )
    unknowns = update(logp, unknowns, grad)
```

```
# Θ(n^3) at every iteration
L = cholesky(σ² * R)
```

General LICM cannot hoist the Cholesky

- σ^2 is a **sampled parameter**, so every iteration updates σ^2

Code you want to write

```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```

```
for sample in range(num_samples):
  for step in range(num_steps):
    logp_fn = logdensity(model)
    logp, grad = value_and_grad(run)(
        logp_fn, inputs, unknowns
    )
    unknowns = update(logp, unknowns, grad)
```

```
# Θ(n^3) at every iteration
L = cholesky(σ² * R)
```

But σ^2 only scales R, so $\text{cholesky}(R)$ can be precomputed

- $\text{cholesky}(\sigma^2 \cdot R) = \sqrt{\sigma^2} \cdot \text{cholesky}(R)$

Code you want to write

```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```

```
for sample in range(num_samples):
  for step in range(num_steps):
    logp_fn = logdensity(model)
    logp, grad = value_and_grad(run)(
        logp_fn, inputs, unknowns
    )
    unknowns = update(logp, unknowns, grad)
```

```
# Θ(n³), precomputed once
L_R = cholesky(R)
```

```
# Θ(n²) at every iteration
~~L = cholesky(σ² * R)~~
L = sqrt(σ²) * L_R
```

Today, experts apply the rewrite by hand

- Requires knowing **library internals**, and either **performance-oriented syntax** or **hand-modified library code**

Code you want to write

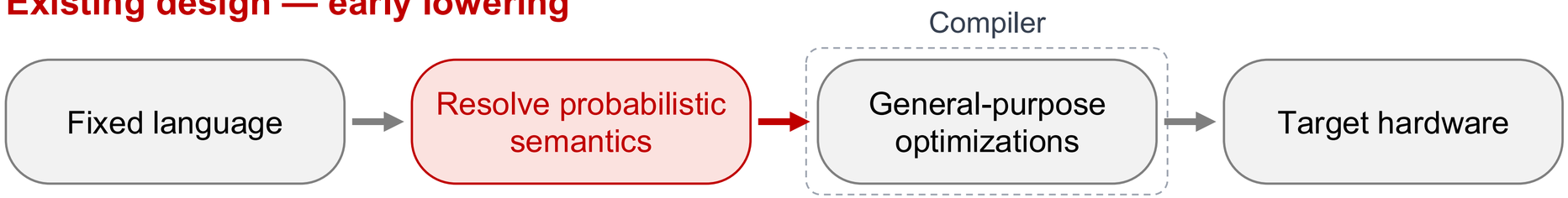
```
function brain_size(R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, σ² * R)  # observed
end
```

Code you want to run

```
L_R = cholesky(R)  # precomputed
function brain_size(L_R, M, G, B) # fixed
  σ² ~ Exponential(1)  # unknown
  a ~ Normal(0, 1)  # unknown
  b_M, b_G ~ Normal(0, 0.5)  # unknown
  μ = a .+ b_M * M .+ b_G * G
  B ~ MvNormal(μ, scale_tril = sqrt(σ²) * L_R)
end
```

Compilers do not recover this work reduction after lowering

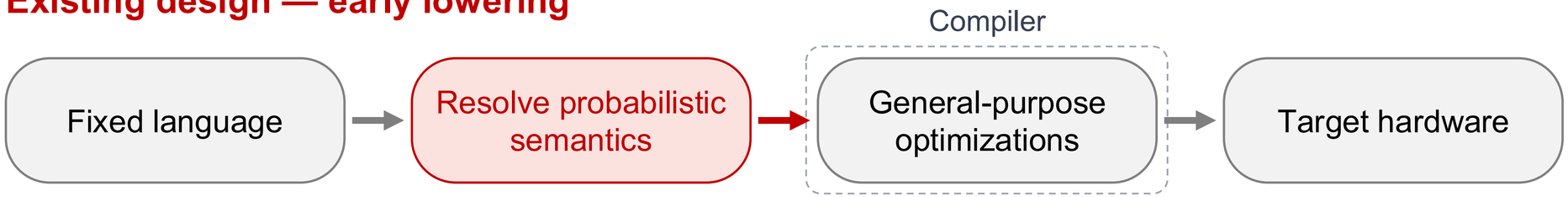
Existing design — early lowering



⚠ Probabilistic semantics lost before code reaches compiler.

Compilers do not recover this work reduction after lowering

Existing design — early lowering



⚠ Probabilistic semantics lost before code reaches compiler.

```
stablehlo.while(%var, %a, %bM, %bG) {
  // per iteration: log-density
  %K = stablehlo.multiply %var, %R
  %L = stablehlo.cholesky %K                    Θ(n^3)
  %z = stablehlo.triangular_solve(%L, %r)
  // gradient code (%X_t = transpose %X)
  %S = stablehlo.dot_general %L, %dL_t
  %Phi = tril(%S) - 0.5 * diag(%S)              Θ(n^3)
  %Y = stablehlo.triangular_solve(%L_t, %Phi)
  %Z = stablehlo.triangular_solve(%L_t, %Y_t)
  %dK = 0.5 * (%Z_t + %Z)
  %d_var = stablehlo.reduce(%dK * %R)
}
```

```
%L_R = stablehlo.cholesky %R
%L_R_t = stablehlo.transpose %L_R  // precomputed
stablehlo.while(%var, %a, %bM, %bG) {
  // per iteration: log-density
  %s = stablehlo.sqrt %var
  %z = stablehlo.triangular_solve(%L_R, %r)
  %u = stablehlo.divide %z, %s
  %q = stablehlo.reduce(%u * %u)
  // gradient code
  %dr = stablehlo.triangular_solve(%L_R_t, %dz)
}
```

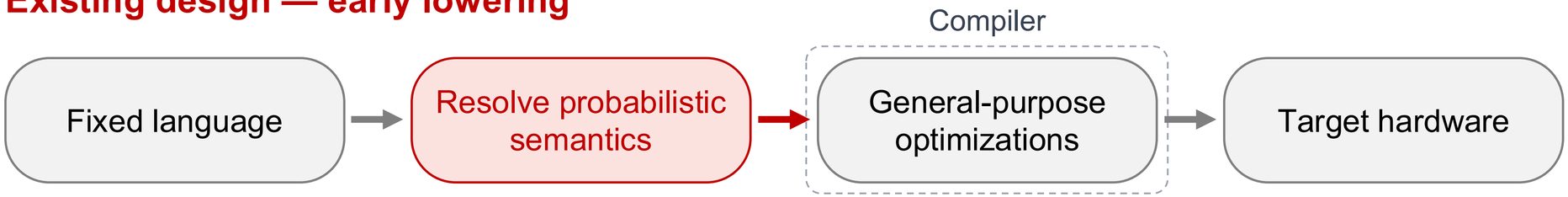
Optimization at a wrong abstraction level; intractable in practice!

Automatically removing redundant work thus requires compiler optimizations at an abstraction level where the **probabilistic semantics** remain explicit.

We propose a compiler-first framework design

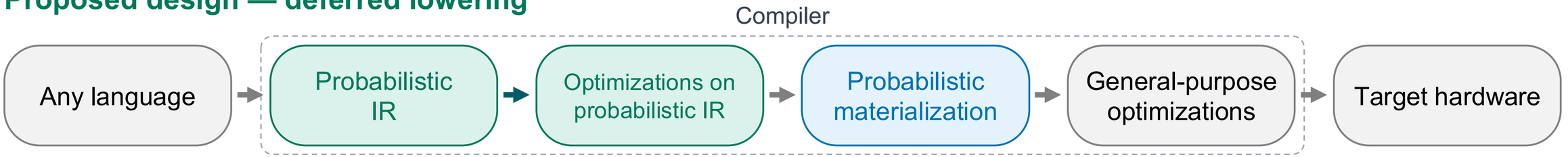
- Give the compiler **first-class probabilistic primitives** and defer the **lowering of probabilistic semantics** into the compiler

Existing design — early lowering



⚠ Probabilistic semantics lost before code reaches compiler.

Proposed design — deferred lowering



✅ Probabilistic semantics preserved by high-level IR.

The Impulse probabilistic programming framework

- **Impulse** realizes the **deferred-lowering** design by making modeling and inference **first-class operations** in the `impulse` dialect

```
// sample a distribution, or call a sub-model
%sigma = impulse.sample @HalfNormal.sampler(%scale)
    <logpdf = @HalfNormal.logpdf, symbol = :sigma>

// forward simulation: build a trace and weight
%trace, %weight, %res = impulse.simulate @model(%args)

// constrained generation: fix observed values
%trace, %weight, %res = impulse.generate @model(%args)
    given %constraints

// inference with an algorithm configuration
%results = impulse.infer @model(%args) given %trace
    <num_warmup = 1000, num_samples = 1000, ...>
```

Principal `impulse` dialect operations

Optimizations run before any inference code exists

1

Model Specification (Any Front End)

σ^2 ~ Exponential(1)
μ = a + b_M * M + b_G * G
B ~ MvNormal(μ, σ^2 * R)

2

Probabilistic IR

```
impulse.infer_region<NUTS> {
  %var = impulse.sample(Exponential)
  %L = impulse.cholesky(%var * %R)
}
```

3

Optimizations on Probabilistic IR

```
%L_R = impulse.cholesky %R   // once
impulse.infer_region<NUTS> {
  %L = math.sqrt(%var) * %L_R
}
```

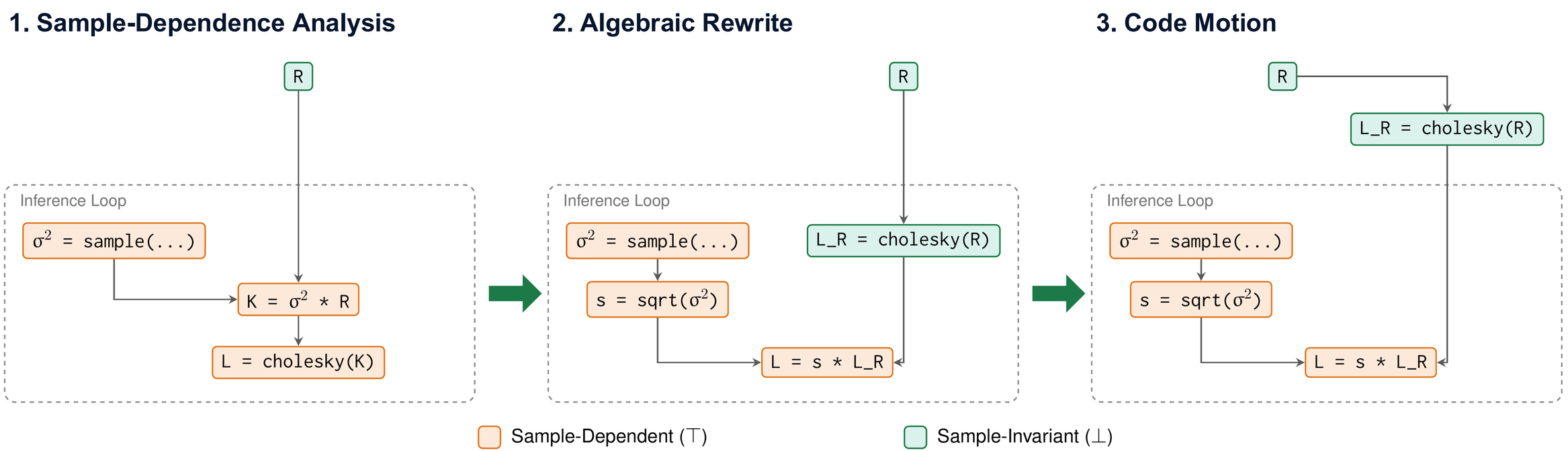
This enables:

1. **Existing standard compiler optimizations** to run directly on the model specification

2. **Novel compiler optimizations** to leverage the probabilistic semantics

Example novel optimization: sample-invariant code motion

- **SICM** algebraically restructures the model specification to prevent **redundant work**



SICM automates manual expert restructuring

- Example SICM rewrite rules, **fully extensible**

CholScale

$$\frac{s : \top \quad A : \bot \quad s \geq 0}{\mathrm{chol}(sA) \rightsquigarrow \sqrt{s}\ \mathrm{chol}(A)}$$

TriSolveScale

$$\frac{s : \top \quad L : \bot}{(sL)^{-1} b \rightsquigarrow s^{-1} L^{-1} b}$$

DotScale

$$\frac{s : \top \quad A : \bot}{(sA)\, x \rightsquigarrow sAx}$$

DotShift

$$\frac{s, t : \top \quad A : \bot}{(sA + tI)\, x \rightsquigarrow sAx + tx}$$

DotAbsorbFFT

$$\frac{x : \top \quad A : \bot \quad F_{jk} = e^{-2\pi \mathrm{i} jk/n}}{A\ \mathrm{FFT}(x) \rightsquigarrow (AF)\, x}$$

DotAbsorbScatter

$$\frac{x : \top \quad A, J : \bot \quad k < n}{A\ \mathrm{scatter}_J(x) \rightsquigarrow A_J\, x}$$

EigenLift

$$\frac{s, t : \top \quad A : \bot \quad (Q, \lambda) = \mathrm{eig}(A)}{(sA + tI)^{-1} b \rightsquigarrow Q\ \mathrm{diag}\left(1/(s\lambda + t)\right) Q^\top b}$$

GaussianSufficientStats

$$\frac{\beta : \top \quad X, y : \bot \quad N > K \quad X = Q_1 R \quad r = \|y - Q_1 Q_1^\top y\|^2}{\|y - X\beta\|^2 \rightsquigarrow \|Q_1^\top y - R\,\beta\|^2 + r}$$

Shaded: fixed work, computed once before inference ⊤: sample-dependent ⊥: sample-invariant

SICM fixes the example with a single rewrite rule

```
impulse.infer_region <algorithm = "NUTS",
                      num_samples = N> {
  %var = impulse.sample(Exponential)
  %a = impulse.sample(Normal)
  %bM, %bG = impulse.sample(Normal)
  %mu = %a + %bM * %M + %bG * %G
  %K = arith.mulf %var, %R
  %L = impulse.cholesky %K
  // observed B
  impulse.sample(MvNormal(%mu, %L))
}
```

```
%L_R = impulse.cholesky %R
impulse.infer_region <algorithm = "NUTS",
                      num_samples = N> {
  %var = impulse.sample(Exponential)
  %a = impulse.sample(Normal)
  %bM, %bG = impulse.sample(Normal)
  %mu = %a + %bM * %M + %bG * %G
  %s = math.sqrt %var
  // observed B
  impulse.sample(MvNormal(%mu, %s*%L_R))
}
```

Optimization at the right abstraction level avoids differentiating Cholesky, and enables an end-to-end speedup of 19.8×

The optimized model is materialized into an inference program

1

Model Specification (Any Front End)

σ^2 ~ Exponential(1)
μ = a + b_M * M + b_G * G
B ~ MvNormal(μ, σ^2 * R)

2

Probabilistic IR

```
impulse.infer_region<NUTS> {
  %var = impulse.sample(Exponential)
  %L = impulse.cholesky(%var * %R)
}
```

3

Optimizations on Probabilistic IR

```
%L_R = impulse.cholesky %R   // once
impulse.infer_region<NUTS> {
  %L = math.sqrt(%var) * %L_R
}
```

4

Probabilistic Materialization

```
impulse.while (%q, %p, %rng) {  // NUTS
  %grad = ...  // gradient code
  // update the unknowns
}
```

Existing compilers perform further optimization and codegen

1

Model Specification (Any Front End)

σ^2 ~ Exponential(1)
μ = a + b_M * M + b_G * G
B ~ MvNormal(μ, σ^2 * R)

2

Probabilistic IR

```
impulse.infer_region<NUTS> {
  %var = impulse.sample(Exponential)
  %L = impulse.cholesky(%var * %R)
}
```

3

Optimizations on Probabilistic IR

```
%L_R = impulse.cholesky %R   // once
impulse.infer_region<NUTS> {
  %L = math.sqrt(%var) * %L_R
}
```

4

Probabilistic Materialization

```
impulse.while (%q, %p, %rng) {  // NUTS
  %grad = ...  // gradient code
  // update the unknowns
}
```

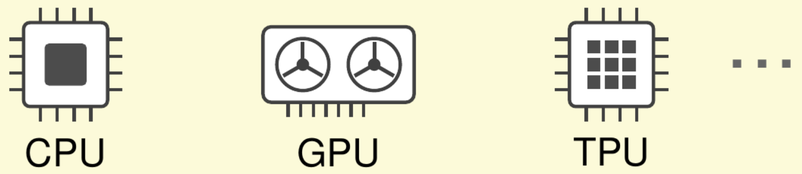
5

Target IR (e.g., StableHLO)

```
stablehlo.while (%state) {
  stablehlo.triangular_solve ...
}
```

6

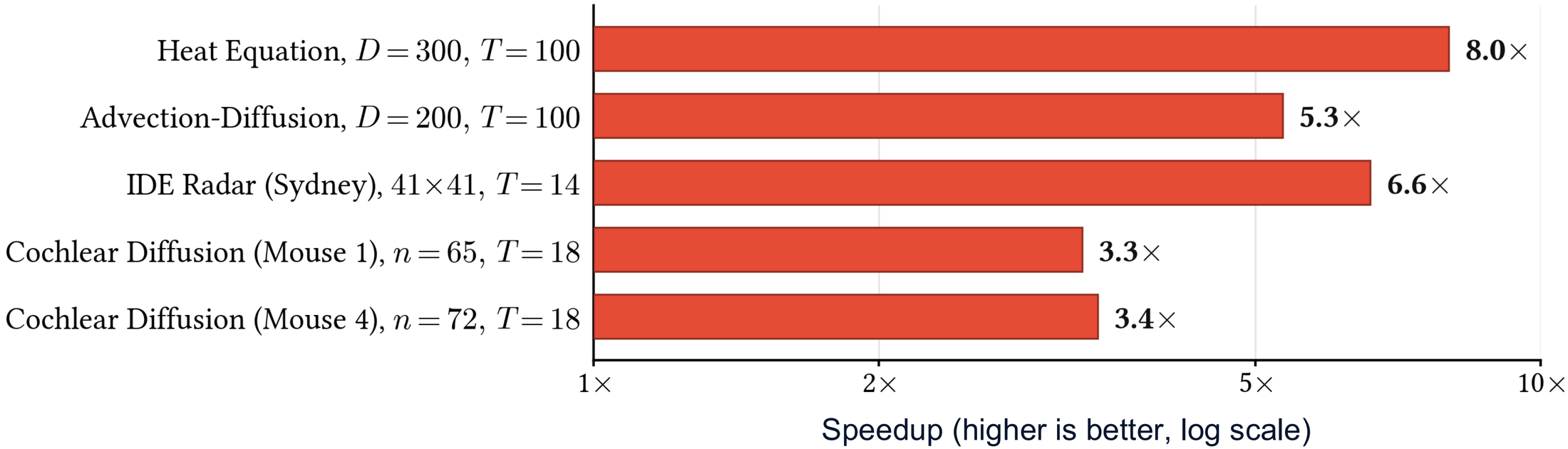
Target Hardware



How much performance are we leaving on the table due to **suboptimal model specifications**?

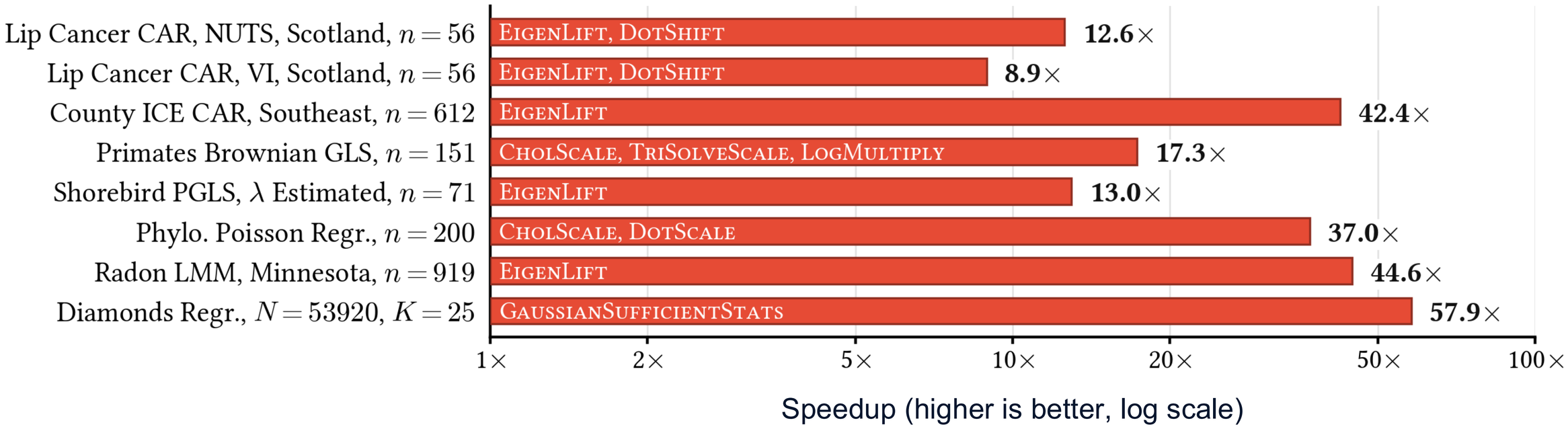
Standard optimizations on the IR: speedups of 3.3× to 8.0×

- **Standard optimizations (e.g., CSE, LICM)** can prevent work invisible from the source, e.g., auto-generated derivatives of expensive operations



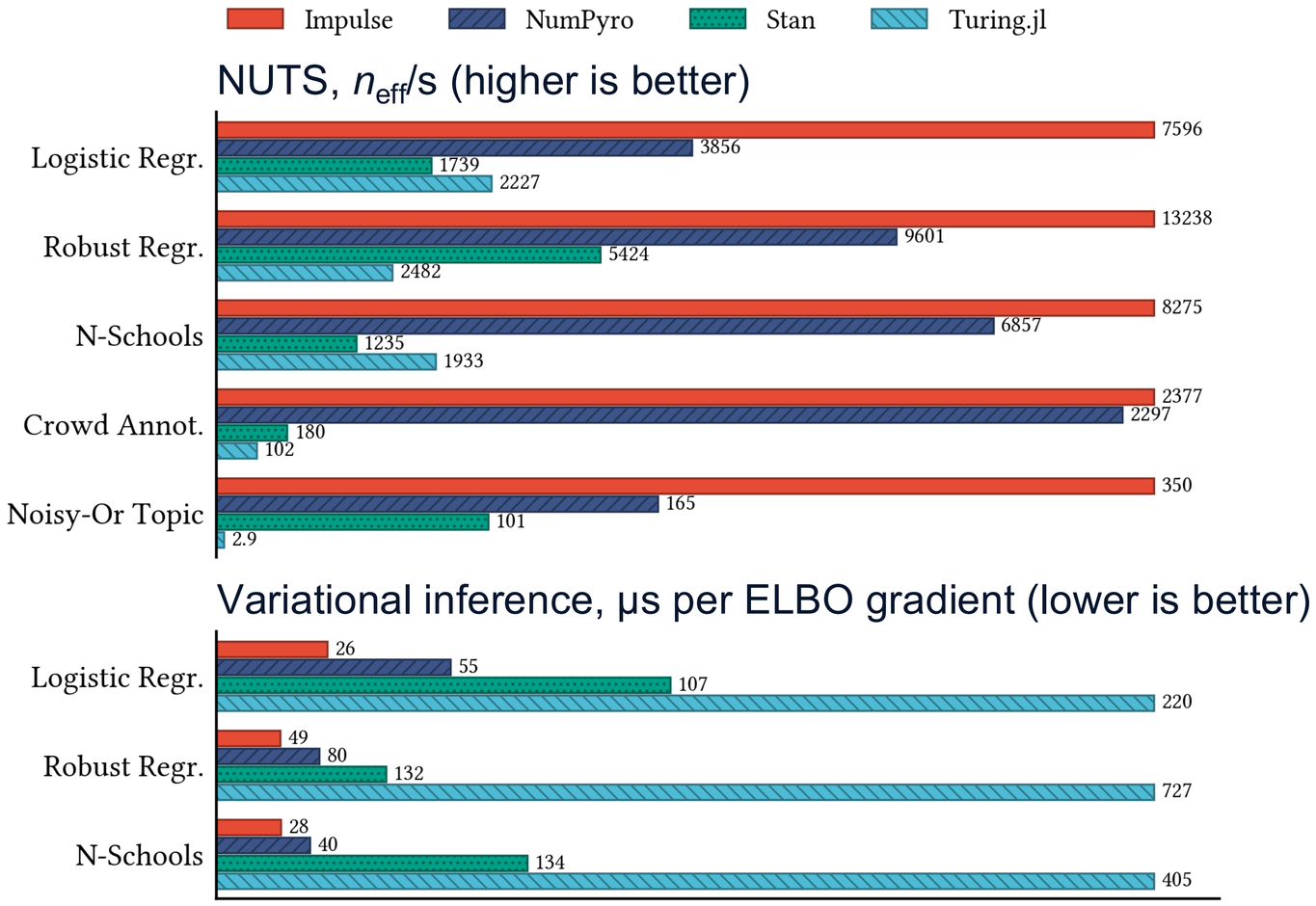
SICM: speedups of 8.9× to 57.9× on top

- Geomean speedup of **23.8×** across 8 real-world models



Compiler-first design delivers competitive performance

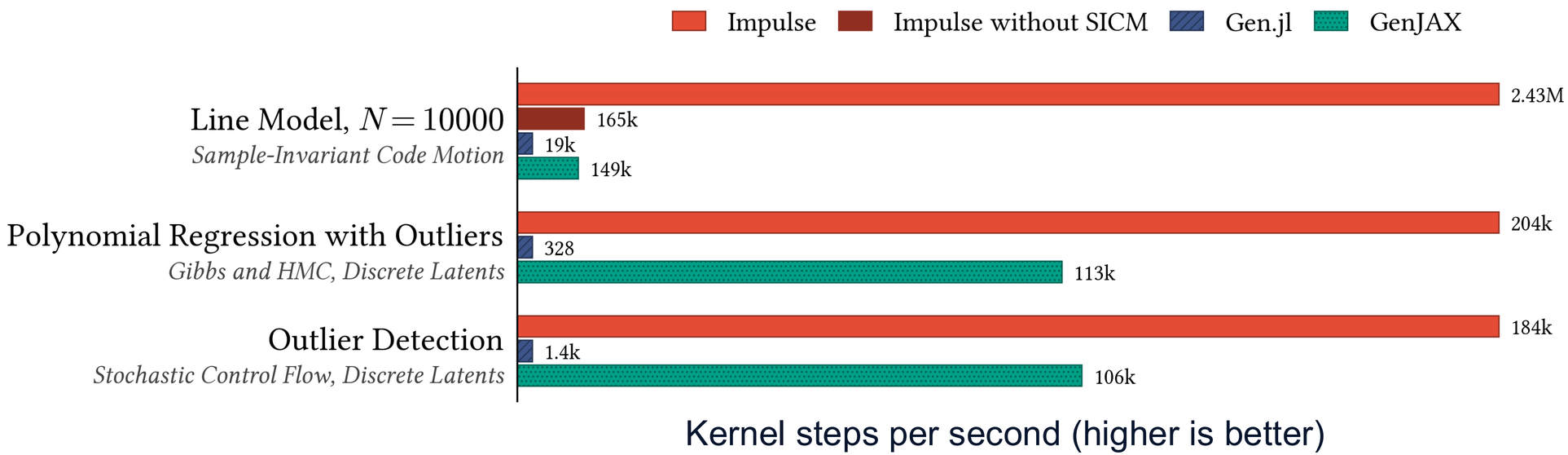
- **PPL Bench**: a suite of expert-written models
- **NUTS**: 1.5×, 5.0×, and 11.7× faster than NumPyro, Stan, and Turing
- **VI**: ELBO gradients 1.7×, 3.8×, and 12.2× faster



Impulse is general across inference paradigms

- NUTS, HMC, VI, and programmable inference kernels are all materialized from the same **probabilistic IR**, so all share **compiler optimizations** on the model specification

- On three programmable inference kernels, Impulse is **3.7×** faster than GenJAX and **218×** faster than Gen.jl



Impulse is general across front-end languages

- Any front end language that emits the **probabilistic IR** benefits from the **same compilation workflow**

Julia

```
_, B = sample(rng, MultivariateNormal(mu, R .* sigma_sq, (n,)); symbol=:B)
```

Python

```
key, B = sample(key, MultivariateNormal(mu, R * sigma_sq, (n,)), "B")
```

C++

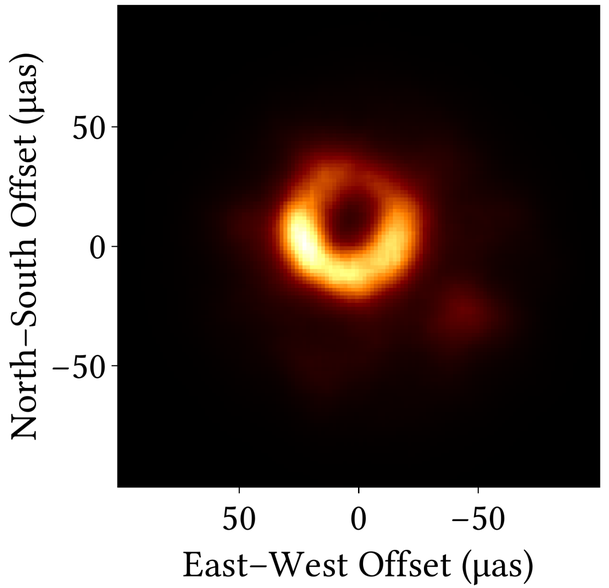
```
auto [k2, B] = sample(k1, MultivariateNormal(mu, R * sigma_sq), "B");
```

Probabilistic IR

```
%K = stablehlo.multiply %R, %s
%B = impulse.sample(%mu, %K)
  <dist = MVN, name = "B">
```

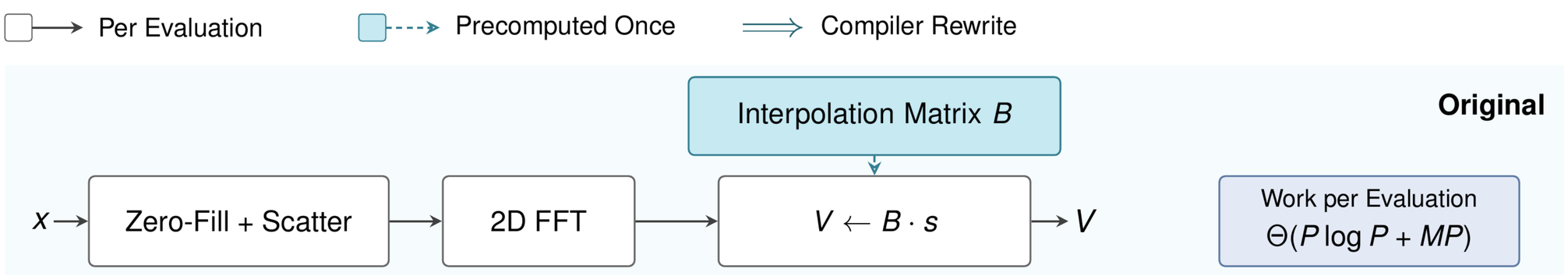
Impulse on a real-world scientific application

- The **Comrade** black-hole imager infers an image of a black hole from Event Horizon Telescope measurements
- Comrade assembles the model from **multiple libraries**, and Impulse inlines the library code into the **probabilistic IR**
- We measure a 15,188-parameter NUTS posterior on GPU



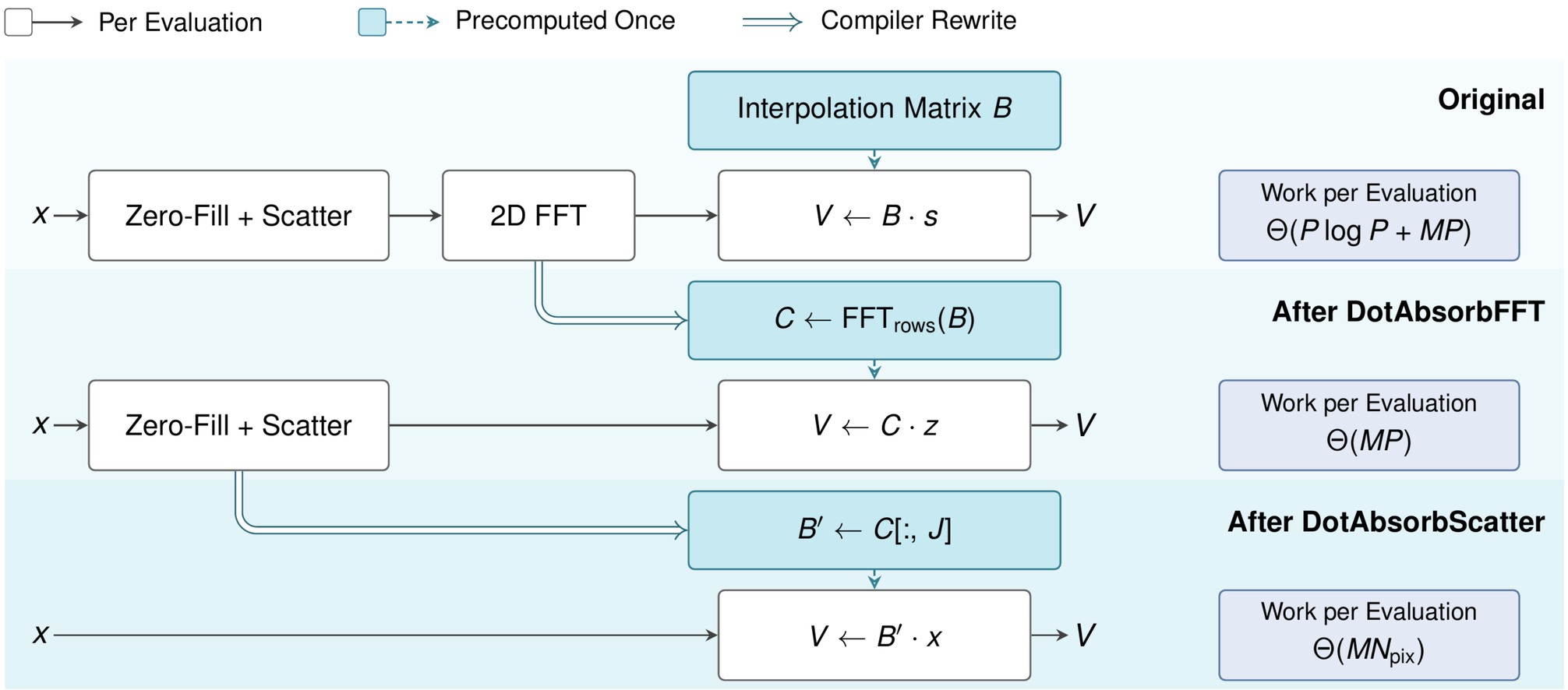
M87* reconstructed by Impulse

SICM removes redundant work that spans libraries



Original program redoes a scatter and an FFT at every evaluation

SICM removes redundant work that spans libraries



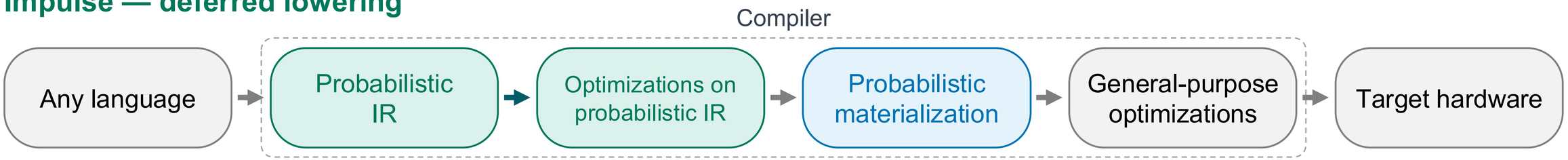
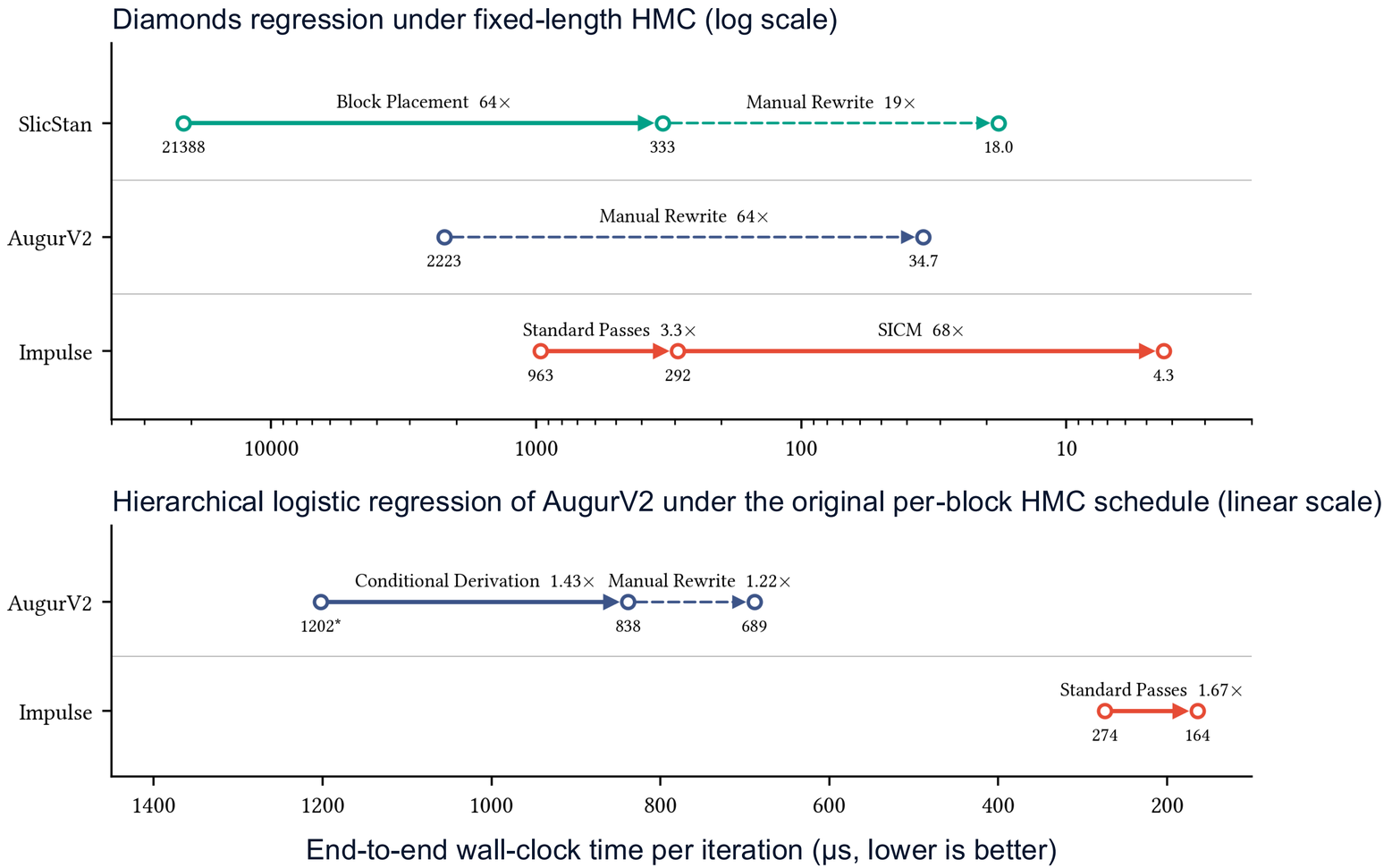
Two SICM rewrites fold the FFT and scatter into a precomputed matrix

SICM enables an end-to-end speedup of 3.7×

Impulse (open source on GitHub: **EnzymeAD/Reactant.jl**)

- Code you want to **write** ≠ code you want to **run**
- **First-class probabilistic primitives** enable optimizations on the **model specification**
- Standard optimizations give speedups of **3.3× to 8.0×**
- SICM gives speedups of **8.9× to 57.9× on top**
- **Competitive**: **1.5×, 5.0×, and 11.7×** faster than NumPyro, Stan, and Turing on PPL Bench
- **Novel optimizations** enable an end-to-end speedup of **3.7×** on real-world astrophysics

Impulse — deferred lowering



Impulse outperforms source-transformation systems