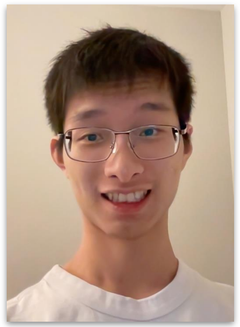
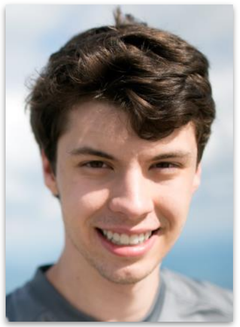
Impulse: Momentously Fast, General, and Portable Probabilistic Programming via Compiler Augmentation





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

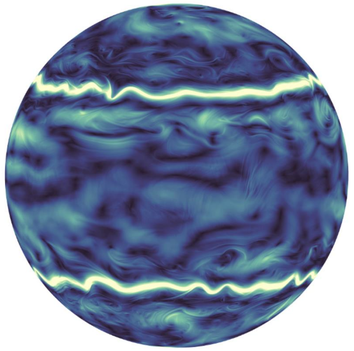
siyuanq4@illinois.edu

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

PLDI Student Research Competition, Boulder, CO
June 18, 2026

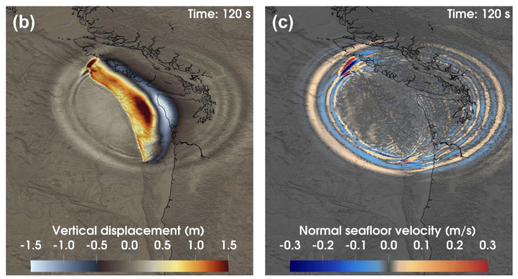
Bayesian Inference + Probabilistic Programming

- **Bayesian inference** infers hidden values from noisy observations by running a **sampler** to explore possibilities
- Bayesian inference is critical when data are limited, indirect, or noisy
- **Probabilistic programming** enables scientists to represent Bayesian statistics as code and run inference algorithms conveniently



Earth Systems Modeling

Credit: https://agupubs.onlinelibrary.wiley.com/doi/pdfdirect/10.1029/2025MS005615



Finite Element Analysis

Credit: https://dl.acm.org/doi/epdf/10.1145/3712285.3771787

Medicine

Credit: https://www.theguardian.com/society/2018/aug/30/modern-medicine-major-threat-public-health

Robotics

Credit: https://foxglove.dev/blog/the-future-of-robotics

Code you want to **write** ≠ code the sampler should **run**.

Code you want to **write** ≠ code the sampler should **run**.

Rather than pushing the burden to users, compilers can enable **fast** inference on **natural** code!

Scientists want to write the model by transcribing formulae

```
function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  K = gp_exp_quad_cov(x, sqrt(α), ρ)
  L = cholesky(K)
  f = L * η
  k ~ Poisson(exp.(f)) # observed
end
```

The sampler runs the model repeatedly within a hot loop!

```
function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  K = gp_exp_quad_cov(x, sqrt(α), ρ)
  L = cholesky(K)
  f = L * η
  k ~ Poisson(exp.(f)) # 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)
```

Inference suffers from repeated deterministic computation!

```
function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  K = gp_exp_quad_cov(x, sqrt(α), ρ)
  L = cholesky(K)
  f = L * η
  k ~ Poisson(exp.(f)) # 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)
```

$\Theta(n^3)$ recomputed at every iteration!

But expensive work only needs to be computed once

```
function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  K = gp_exp_quad_cov(x, sqrt(α), ρ)
  L = cholesky(K)
  f = L * η
  k ~ Poisson(exp.(f)) # observed
end
```

```
...
  R = covariance(x, ρ) # loop-invariant
  L = cholesky(α · R) # loop-variant
...
```

$$\text{cholesky}(\alpha \cdot R) = \text{sqrt}(\alpha) \cdot \text{cholesky}(R)$$

```
...
  R = covariance(x, ρ)
  LR = cholesky(R)   # loop-invariant
  L = sqrt(α) · LR   # loop-variant
...
```

Today, users must rewrite the model by hand

```
function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  K = gp_exp_quad_cov(x, sqrt(α), ρ)
  L = cholesky(K)
  f = L * η
  k ~ Poisson(exp.(f)) # observed
end
```

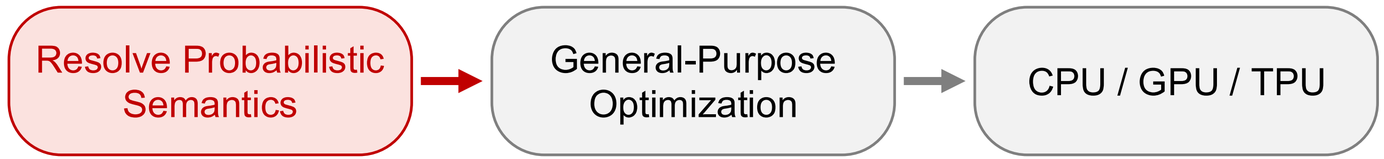
```
K = gp_exp_quad_cov(x, 1, ρ)
LR = cholesky(R) # precomputed

function model(x, k, ρ) # fixed
  n = length(x)
  α ~ HalfNormal(2) # unknown
  η ~ Normal(0, 1, (n,)) # unknown
  f = (sqrt(α) * LR) * η
  k ~ Poisson(exp.(f)) # observed
end
```

⚠ Expert analysis required to identify this redundancy!

Why can’t existing frameworks fix this redundancy?

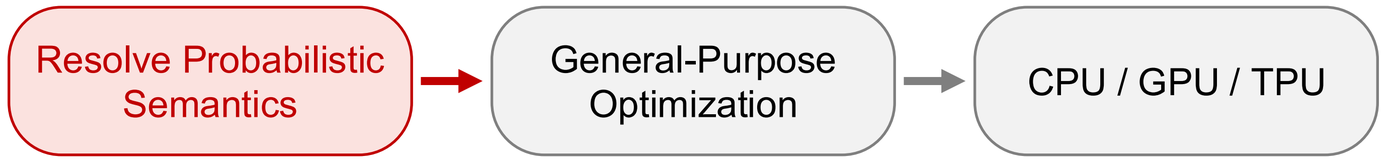
Existing – Early Lowering



⚠ Probabilistic structure lost before code reaches compiler.

Why can’t existing frameworks fix this redundancy?

Existing – Early Lowering



⚠ Probabilistic structure lost before code reaches compiler.

```
stablehlo.while { // per leapfrog step
  // log-density evaluation
  %K = stablehlo.multiply %alpha_sq, %R
  %L = stablehlo.cholesky %K
  %f = stablehlo.dot_general %L, %eta
  // adjoint
  %S = stablehlo.dot_general %L, %dL_t
  %Phi = tril(%S) + 0.5 * diag(%S)
  %Y = stablehlo.triangular_solve(%L_t, %Phi)
  %Z = stablehlo.triangular_solve(%L_t, %Y_t)
  %dA = 0.5 * (%Z_t + %Z)
  %d_alpha = stablehlo.reduce(%dA * %R)
}
```

```
%L_R = stablehlo.cholesky %R
stablehlo.while { // per leapfrog step
  // log-density evaluation
  %s = stablehlo.sqrt %alpha_sq
  %v = stablehlo.dot_general %L_R, %eta
  %f = stablehlo.multiply %s, %v
  // adjoint
  %d_alpha = stablehlo.reduce(%df * %v)
  %d_eta = stablehlo.dot_general %alpha_df, %L_R_t
}
```

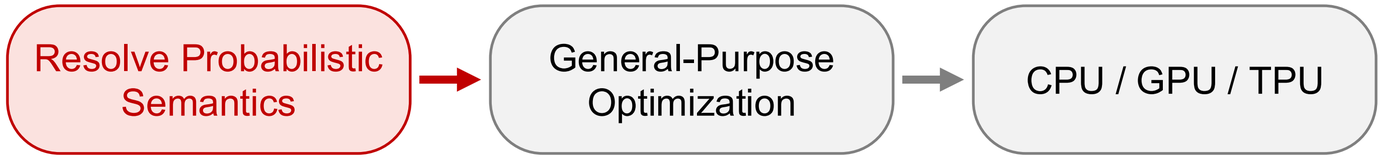
Optimization at a wrong abstraction level; intractable in practice!

Can we preserve probabilistic structure in compiler IR?

Can a compiler automatically enable **fast** inference from **natural** code?

Solution: First-Class Probabilistic IR within the Compiler

Existing – Early Lowering



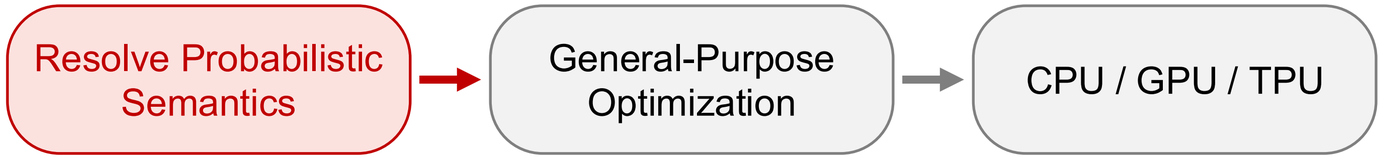
⚠ Probabilistic structure lost before code reaches compiler.

Impulse — Deferred Lowering

✅ Probabilistic structure preserved by high-level IR

Solution: First-Class Probabilistic IR within the Compiler

Existing – Early Lowering



⚠ Probabilistic structure lost before code reaches compiler.

Impulse — Deferred Lowering

✅ Probabilistic structure preserved by high-level IR

First-class IR enables 1) existing optimizations to run on probabilistic code and 2) novel probabilistic optimizations!

Sample-Invariant Code Motion (SICM)

Impulse — Deferred Lowering

✅ Probabilistic structure preserved by high-level IR, enabling new compiler optimizations!

- **Sample-dependence analysis** identifies repeated deterministic computation

```
...
  R = covariance(x, ρ)
  L = cholesky(α · R)
...
```

```
...
  R = covariance(x, ρ) # Sample-invariant
  L = cholesky(α · R)  # Sample-dependent
...
```

Sample-Invariant Code Motion (SICM)

Impulse — Deferred Lowering

✅ Probabilistic structure preserved by high-level IR, enabling new compiler optimizations!

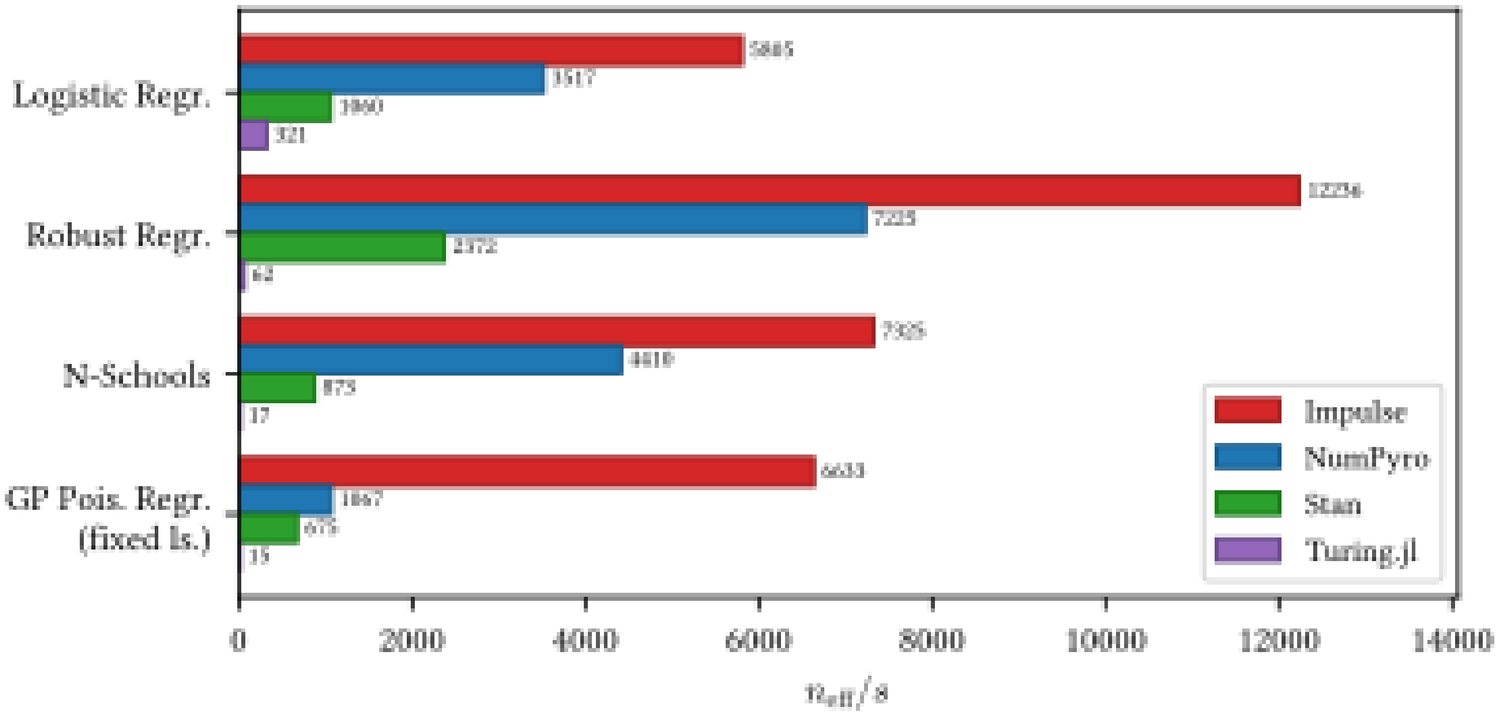
- **Sample-dependence analysis** identifies repeated deterministic computation
- SICM rewrite patterns produce hoistable expensive operations

```
...
  R = covariance(x, ρ)
  L = cholesky(α · R)
...
```

```
...
  R = covariance(x, ρ)
  LR = cholesky(R) # Hoisted
  L = sqrt(α) · LR
...
```

Evaluation

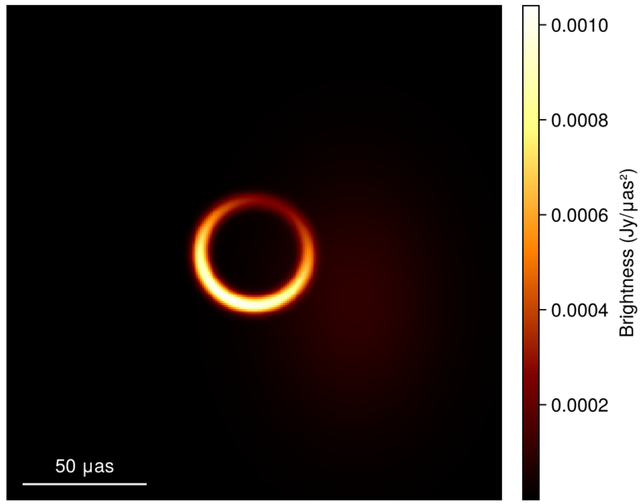
- Standard compiler passes on high-level IR: **8× – 36×**
- \+ Probabilistic-specific optimizations: **8× – 190×**
- On PPLBench
  - **1.7×** faster than NumPyro
  - **6.4×** faster than Stan
  - **53×** faster than Turing.jl



Higher is better.

Real Workload: Comrade Black Hole Imager

- Inverse problem: infer an image of a black hole from telescope measurements
- 4,428-parameter NUTS posterior on GPU
- Impulse folds an FFT hidden behind multiple layers of library calls into one precomputed matrix product
- **3.7× end-to-end speedup from probabilistic opts**



Credit: [ptiede.github.io/Comrade.jl — GeometricModeling tutorial]

Impulse (Open Source on GitHub: **EnzymeAD/Reactant.jl**)

- Code you want to write ≠ code the sampler should run.
- Impulse keeps probabilistic structure visible to the compiler
- Combining with new and existing optimizations enable order-of-magnitude speedups
- Ablation test of novel probabilistic optimizations enable **3.7× speedup** on state-of-the-art GPU black hole imaging

Impulse — Deferred Lowering

Example High-Level Probabilistic IR

```
impulse.infer_region
    <{alg = "NUTS", num_samples = N}> {
  %alpha = impulse.sample(HalfNormal)
  %eta = impulse.sample(Normal)
  %R = stablehlo.exponential @f(%x, %rho)
  %K = stablehlo.multiply %alpha_sq, %R
  %L = stablehlo.cholesky %K
  %f = stablehlo.dot_general %L, %eta
  %rate = stablehlo.exponential %f
  impulse.sample(Poisson(%rate))
}
```

```
%L_R = stablehlo.cholesky %R
impulse.infer_region
    <{alg = "NUTS", num_samples = N}> {
  %alpha = impulse.sample(HalfNormal)
  %eta = impulse.sample(Normal)
  %s = stablehlo.sqrt %alpha_sq
  %v = stablehlo.dot_general %L_R, %eta
  %f = stablehlo.multiply %s, %v
  %rate = stablehlo.exponential %f
  impulse.sample(Poisson(%rate))
}
```