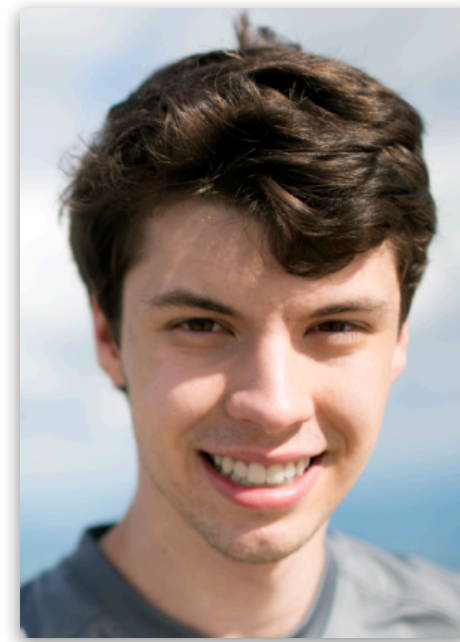
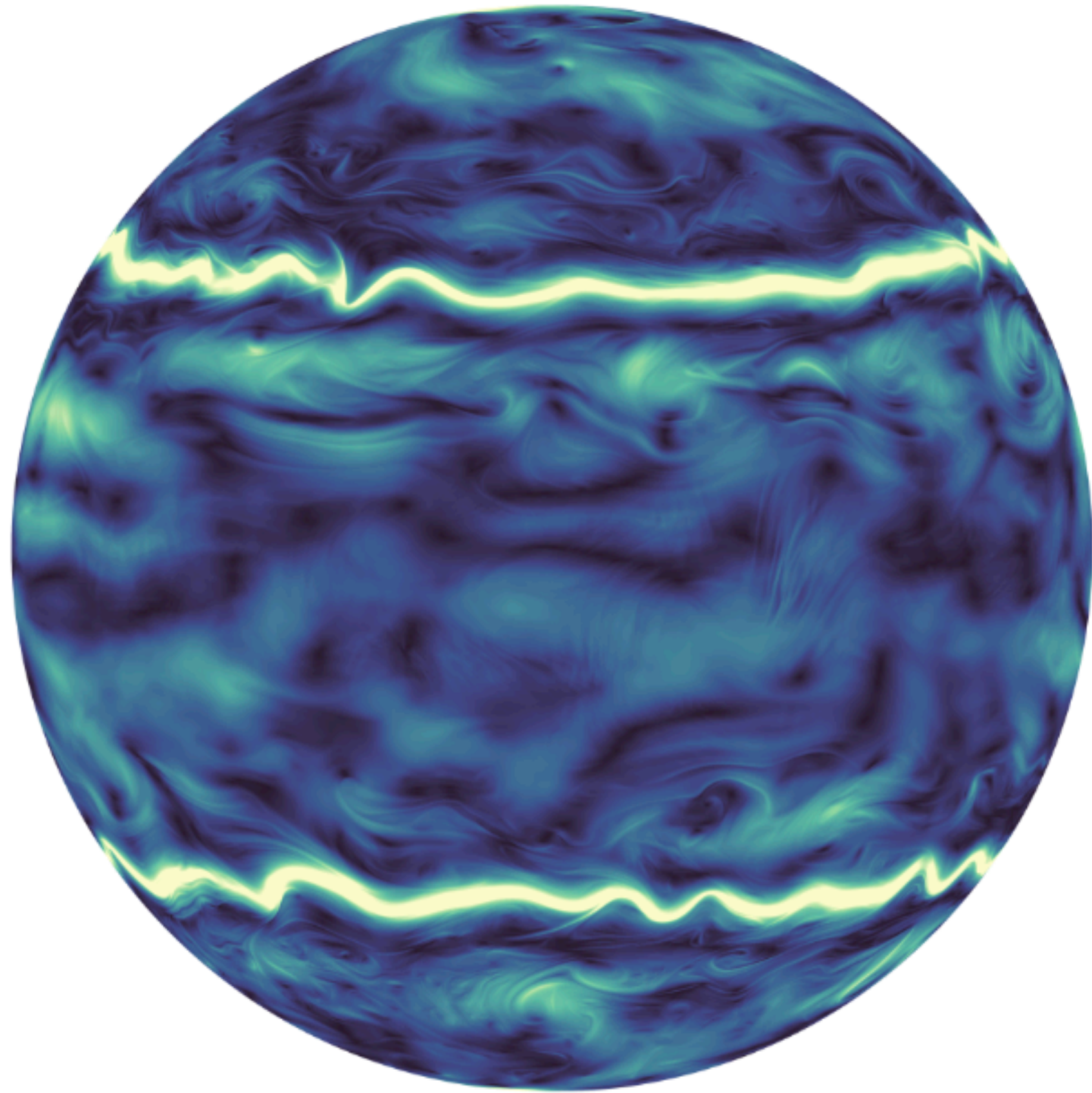
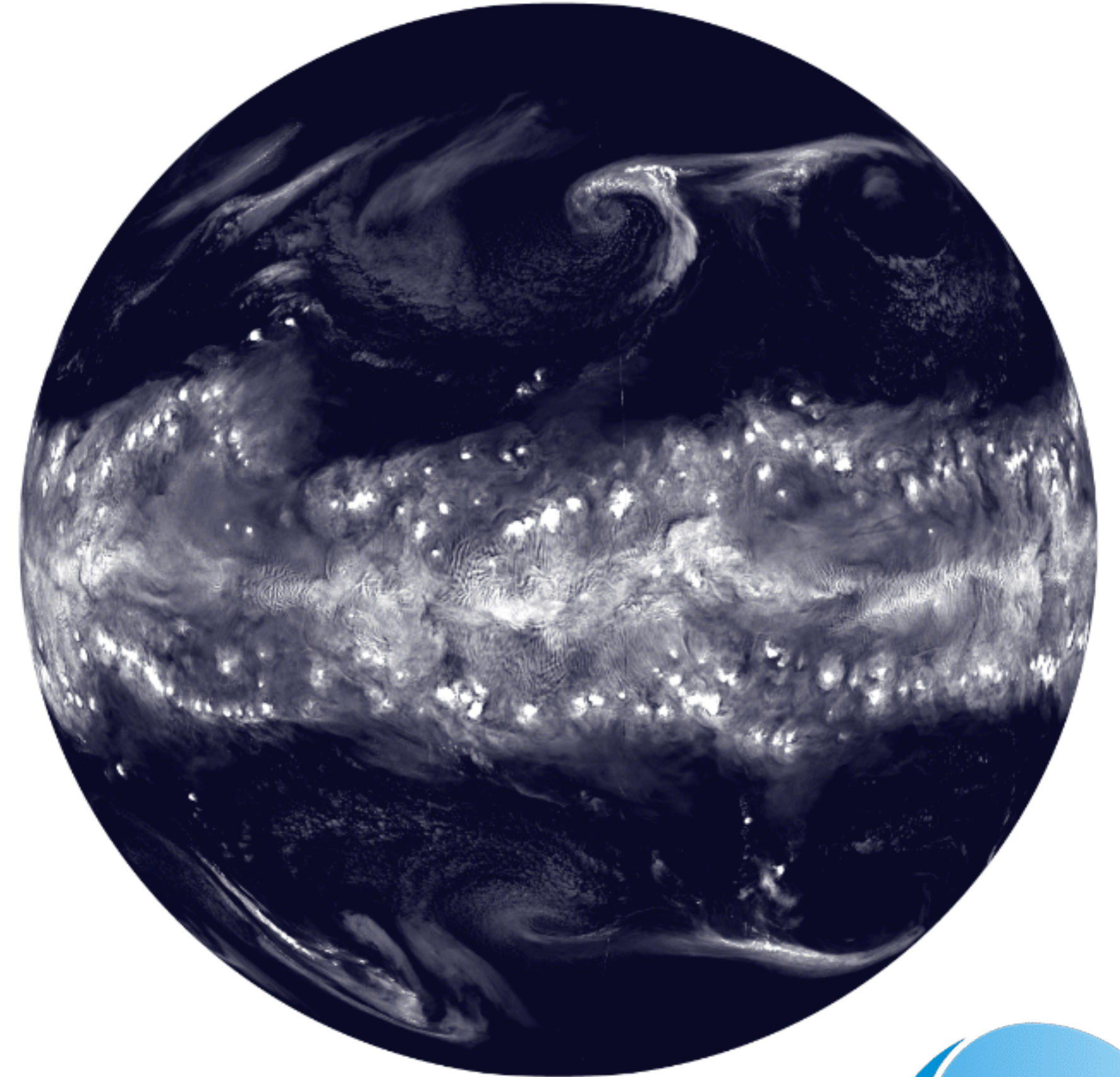


A Compiler-First Planetary Compute Engine: Automatic differentiable and performance portable Earth System Modeling



William S. Moses



wsmoses@illinois.edu

PROPL '26

June 15, 2026



UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN



Key Points:

- Four Earth system model components are successfully differentiated using the reverse mode of the automatic differentiation tool Enzyme
- The Julia-based, graphics processing unit-enabled models use bespoke numerics, with finite-volume, finite-element, and spectral spatial discretization schemes
- The compiler transpilation tool Reactant enables optimized, portable performance across diverse ML-customized HPC architectures

Supporting Information:

Supporting Information may be found in the online version of this article.

Correspondence to:

P. Heimbach,
heimbach@utexas.edu

Citation:

Moses, W. S., Cheng, G., Churavy, V., Gelbrecht, M., Klöwer, M., Kump, J., et al. (2026). DJ4Earth: Differentiable, and performance-portable Earth system modeling via program transformations. *Journal of Advances in Modeling Earth Systems*, 18, e2025MS005615. <https://doi.org/10.1029/2025MS005615>

Received 12 NOV 2025
Accepted 20 APR 2026

Author Contributions:

Conceptualization: Mathieu Morlighem, Christopher Hill, Nora Loose, Sri Hari Krishna Narayanan, Patrick Heimbach
Formal analysis: William S. Moses, Gong Cheng, Valentin Churavy, Maximilian Gelbrecht, Milan Klöwer, Joseph Kump, Mathieu Morlighem, Sarah Williamson, Dhruv Apte, Nora Loose, Sri Hari Krishna Narayanan, Michel Schanen, Greg Wagner, Patrick Heimbach

© 2026 The Author(s). Journal of Advances in Modeling Earth Systems published by Wiley Periodicals LLC on behalf of American Geophysical Union. This is an open access article under the terms of the [Creative Commons Attribution License](#), which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

DJ4Earth: Differentiable, and Performance-Portable Earth System Modeling via Program Transformations

William S. Moses¹, Gong Cheng², Valentin Churavy³, Maximilian Gelbrecht⁴, Milan Klöwer⁵, Joseph Kump⁶, Mathieu Morlighem², Sarah Williamson⁶, Dhruv Apte⁶, Paul Berg⁷, Mosè Giordano⁸, Christopher Hill⁹, Nora Loose¹⁰, Alexis Montoisson¹¹, Sri Hari Krishna Narayanan¹¹, Avik Pal⁹, Michel Schanen¹¹, Simone Silvestri^{9,12}, Greg Wagner⁹, and Patrick Heimbach⁶

¹University of Illinois Urbana-Champaign, Champaign, IL, USA, ²Dartmouth College, Hanover, NH, USA, ³Johannes Gutenberg University Mainz & University of Augsburg, Mainz, Germany, ⁴Technical University of Munich & Potsdam Institute for Climate Impact Research, Potsdam, Germany, ⁵University of Oxford, Oxford, UK, ⁶University of Texas at Austin, Austin, TX, USA, ⁷Bern University of Applied Sciences, Bern, Switzerland, ⁸University College London, London, UK, ⁹Massachusetts Institute of Technology, Cambridge, MA, USA, ¹⁰CJWorthy, LLC, Boulder, CO, USA, ¹¹Argonne National Laboratory, Lemont, IL, USA, ¹²Politecnico di Torino, Torino, Italy

Abstract Differentiable Earth system models (ESMs) enable powerful applications such as sensitivity analysis, gradient-based calibration, state estimation, boundary flux inversions, uncertainty quantification, and online machine learning. Reverse-mode automatic differentiation (AD) efficiently provides gradients for such tasks, yet models have rarely included this capability because of complex, bespoke numerical algorithms. As part of the *Differentiable programming in Julia for Earth system modeling (DJ4Earth)* initiative, we present improved capabilities of the AD tool Enzyme.jl and the new compiler transpilation tool Reactant.jl, augmented by sophisticated checkpointing algorithms, which, together make general-purpose AD tractable and efficient for full-fledged ESM components written in Julia. Operating at the low-level virtual machine intermediate representation or multi-level intermediate representation compiler levels, these frameworks support mutable memory, custom kernels, and compiler optimizations before and after differentiation. Julia-specific challenges related to just-in-time compilation and garbage collection are handled efficiently. Reactant further enables automatic performance portability across central processing units, graphics processing units, and tensor processing units, facilitating use of emerging AI-customized high-performance computing architectures. We demonstrate these frameworks on four Julia-based ESM components featuring diverse spatial discretizations and numerical algorithms: the rotating-sphere shallow water model ShallowWaters.jl, the finite-volume ocean model Oceananigans.jl, the finite-element ice sheet model DJUICE.jl, and the spectral atmospheric model SpeedyWeather.jl. Across these ESM components, our tools compute efficient and correct gradients. These results establish a foundation for differentiable, high-performance and performance-portable ESMs that can integrate neural networks for unresolved processes, trained online, enabling next-generation hybrid physics-machine learning ESMs constrained by physical dynamics and observations.

Plain Language Summary Earth system models are computer programs that simulate how Earth's atmosphere, ocean, ice, and biosphere interact and evolve. These models consist of millions of lines of code and rely on uncertain inputs. To improve accuracy, scientists adjust these inputs to minimize the difference between simulations and observations, measured by a “cost function.” Another computer program can efficiently determine how changes in each input affect the outcome. This calculation, called the gradient of the cost function, would be extremely time-consuming to code manually. Instead, we use an automatic differentiation (AD) tool called Enzyme, which computes these gradients efficiently and updates them automatically whenever the model changes. As computing systems evolve rapidly, especially those optimized for artificial intelligence (AI), another tool called Reactant enables models to run efficiently across different hardware, from central processing units to graphics processing units and AI accelerators. We demonstrate these methods on four Earth system model components written in the modern programming language Julia: a shallow water model, an ocean model, an ice sheet model, and an atmospheric model. For each, the code generated via AD produces correct gradients of the cost function. This work lays the foundation for combining these differentiated models with machine learning to improve model accuracy efficiently.

Making Waves in the Cloud: A Paradigm Shift for Atmosphere and Ocean Modeling through Compiler Technology

William S. Moses^{†§}, Mosè Giordano^{*}, Avik Pal[‡], Gregory Wagner^{‡¶}, Daniel Kyte-Zable^{□¶}, Ivan R. Ivanov^{††}, Paul Berg[▽], Johannes Blaschke, Roman Lee[▽], Jules Merckx[△], Arpit Jaiswal[△], Joseph Kump[#], Patrick Heimbach[#], Rosie Zou[§], Tongfei Guo[§], Son Vu^{‡b}, Sergio Sanchez-Ramirez[◇], Simone Silvestri, Nora Loose^{*}, Ivan Ho, Vimarsh Sathia[‡], Jan Hueckelheim^{*}, Johannes De Fine Licht, Kevin Gleason[§], Ludovic Räss^{◇◇}, Gabriel Baraldi, Dhruv Apte[#], Tobias Bischoff[¶], Lorenzo Chelini[△], Jacques Pienaar[§], Gaetan Lounes, Navid Constantinou, William R. Magro[§], Michel Schanen[△], Alexis Montoisson[△], Alan Edelman[‡], Samarth Narang, Tobias Grosser, Keno Fischer[‡], Valentin Churavy[#], Robert Hundt[§], Albert Cohen[§], Oleksandr Zinenko^{‡b}

UIUC[†], Google[§], UCL^{*}, MIT[‡], NVIDIA[△], UT Austin^{*}, Brium^{‡b}, [CJ]Worthy[△], BSC[◇], Argonne National Laboratory[△], Lawrence Berkeley National Laboratory[▽], Cambridge^b, JuliaHub^b, Johannes Gutenberg University, Mainz & University of Augsburg[#], BFH[▽], Ghent University[△], Aeolus Labs[‡], Brown University[□], University of Lausanne^{◇◇}, Institute of Science Tokyo^{††}

Abstract

Earth system simulation is limited by HPC compute availability rather than algorithms, requiring scientists and decision-makers to balance fidelity against high costs. Yet exaFLOPs of cloud infrastructure compute remain underused for scientific computation due to technological differences: code written in C++ or Julia for traditional MPI-over-Infiniband systems translates poorly to AI-oriented low-precision accelerators with custom programming interfaces. Even supercomputers integrate AI-oriented accelerators, making it difficult to achieve peak performance without porting software to an AI framework. This work proposes a fully automated cross-compilation and automatic differentiation process for scientific simulation—demonstrated via global-scale Julia-based ocean and atmosphere forward and adjoint models—transparently across supercomputers with GPU accelerators and the Google Cloud TPU AI-oriented platform. It automatically reduces inter-node communication—the main scaling bottleneck—and applies algebraic transformations to leverage AI hardware. This work unlocks cloud-scale AI hardware, and the massive industry investment behind it, for the most demanding problems in computational climate science.

*Correspondence: wsmoses@illinois.edu

Author's Contact Information: William S. Moses^{†§}, Mosè Giordano^{*}, Avik Pal[‡], Gregory Wagner^{‡¶}, Daniel Kyte-Zable^{□¶}, Ivan R. Ivanov^{††}, Paul Berg[▽], Johannes Blaschke, Roman Lee[▽], Jules Merckx[△], Arpit Jaiswal[△], Joseph Kump[#], Patrick Heimbach[#], Rosie Zou[§], Tongfei Guo[§], Son Vu^{‡b}, Sergio Sanchez-Ramirez[◇], Simone Silvestri, Nora Loose^{*}, Ivan Ho, Vimarsh Sathia[‡], Jan Hueckelheim^{*}, Johannes De Fine Licht, Kevin Gleason[§], Ludovic Räss^{◇◇}, Gabriel Baraldi, Dhruv Apte[#], Tobias Bischoff[¶], Lorenzo Chelini[△], Jacques Pienaar[§], Gaetan Lounes, Navid Constantinou, William R. Magro[§], Michel Schanen[△], Alexis Montoisson[△], Alan Edelman[‡], Samarth Narang, Tobias Grosser, Keno Fischer[‡], Valentin Churavy[#], Robert Hundt[§], Albert Cohen[§], Oleksandr Zinenko^{‡b}, UIUC[†], Google[§], UCL^{*}, MIT[‡], NVIDIA[△], UT Austin^{*}, Brium^{‡b}, [CJ]Worthy[△], BSC[◇], Argonne National Laboratory[△], Lawrence Berkeley National Laboratory[▽], Cambridge^b, JuliaHub^b, Johannes Gutenberg University, Mainz & University of Augsburg[#], BFH[▽], Ghent University[△], Aeolus Labs[‡], Brown University[□], University of Lausanne^{◇◇}, Institute of Science Tokyo^{††}.

1 Justification for ACM Gordon Bell Prize for Climate Modeling

This work enables seamless performance portability for scientific code, demonstrated by re-targeting and accelerating Julia and CUDA-based ocean general circulation models across supercomputers and Google Cloud TPU platform—offering compute capacity exceeding supercomputers—using automated compiler technology to target an AI-oriented accelerator programming interface, achieving ideal scaling efficiency $\geq 100\%$.

2 Performance Attributes

Performance Attribute	This Submission
Category achievement	scalability
Type of method used	explicit
Results reported on the basis of precision reported	whole application except I/O single and double precision (GPU) emulated single and double precision (TPU)
System scale	results measured on full-scale system
Measurement mechanism	timers, FLOP count

3 Overview of the Problem

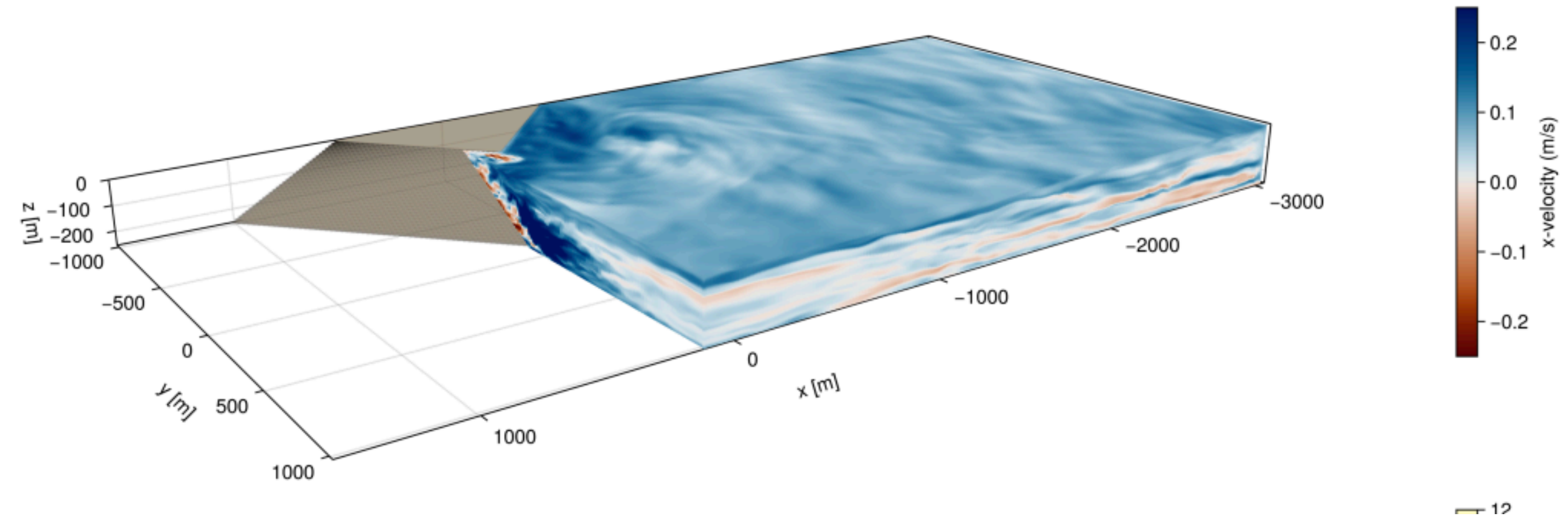
Climate is governed by geophysical fluid dynamics. This submission focuses on the core of global climate models: simulations of the fluid dynamics of the ocean and atmosphere that dictate the large-scale structure and long-term evolution of Earth's climate (Fig. 1). Fluid dynamical processes underpin oceanic phenomena such as equator-to-pole heat transport, the westward-intensified wind-driven gyre circulation, the buoyancy-driven thermohaline circulation, coupled atmosphere-ocean processes such as El Niño, mesoscale eddies, and turbulent mixing, all of which interact, drive variability and set the climate's memory and predictability [15, 24, 58]. Accurately simulating climate requires ocean and atmospheric dynamical cores to solve the governing equations of fluid motion on the rotating sphere as efficiently and accurately as possible on given computational resources.

DJ4Earth Collaboration
Journal of Atmospheric Modeling
May 2026

Reactant Collaboration
In Submission to
Gordon Bell for Climate Science

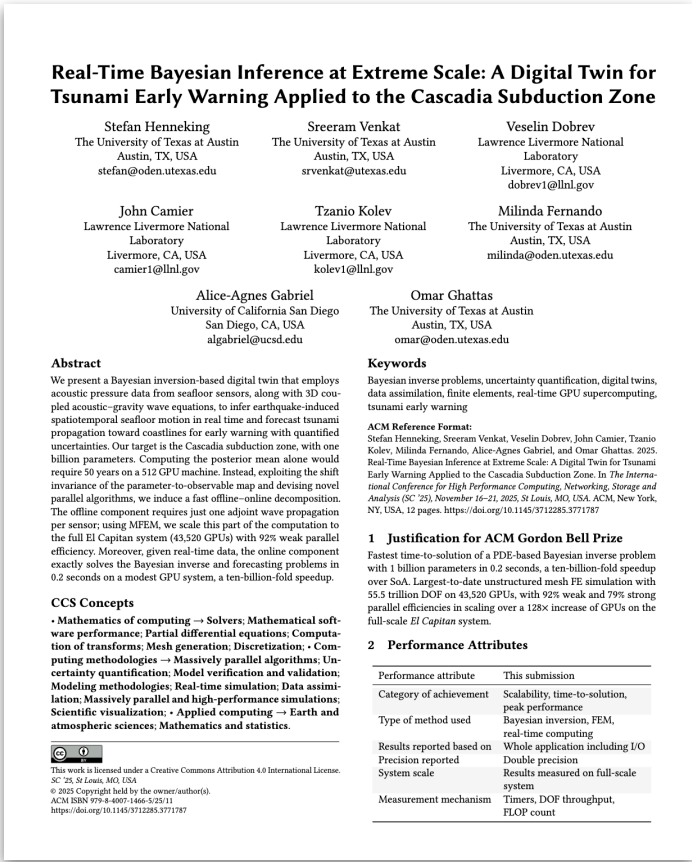
Challenges Behind Modern Day Earth Systems: Scale

- Simulation requires:
 - Physics at a variety of levels
 - Increasing detail (resolution) and complexity (process representation)
 - Creates vast output (e.g., 10-100TB/simulation, ~10PB for CMIP6)
 - Arctic simulation at 1-2 km resolution takes >7 million core hours, 1.1PB output, running on ~10,000 cores
- “Pancake Earth”
 - High resolution in X and Y axis, not much Z

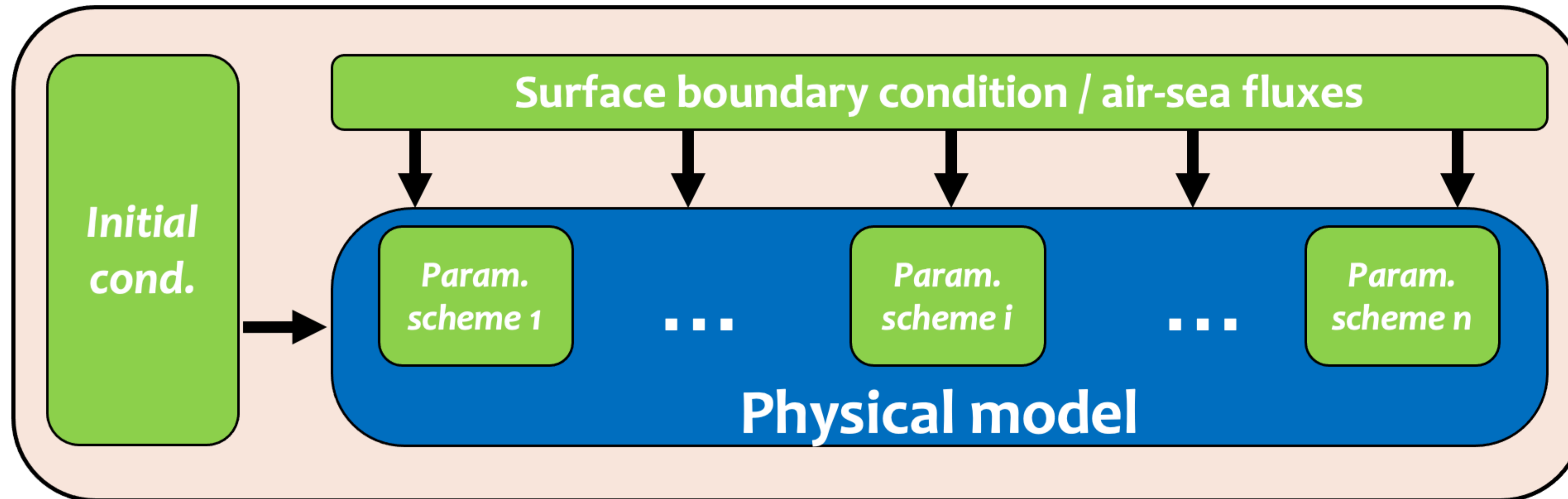


Challenges Behind Modern Day Earth Systems: Data

- Real-world data is extremely sparse and heterogeneous.
- Given both incomplete observations and simulations, how can we derive useful results, like
 - Casual, dynamical attribution
 - Detection of small, residual signals in noisy system
 - Computing comprehensive uncertainties
 - Informing efficient observing strategies

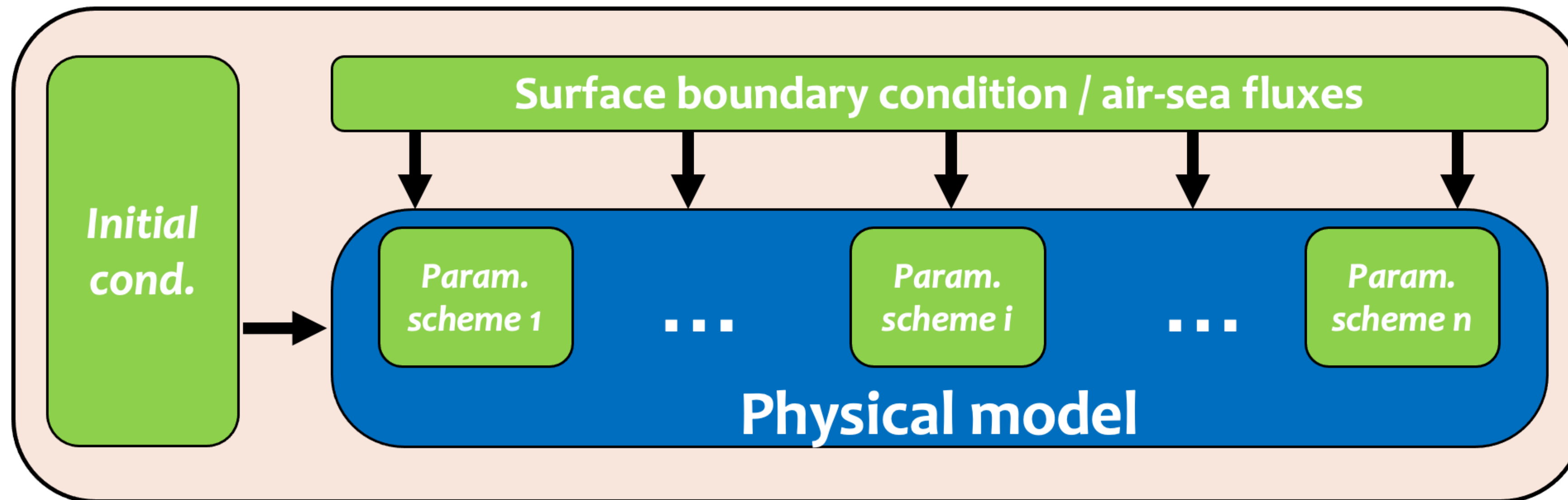


Challenges Behind Modern Day Earth Systems: Models



- Significant uncertainties remain, including:
 - Uncertain inputs (initial and boundary conditions)
 - Parametric and structural model uncertainties (requiring calibration)
 - Remote influences limiting regional simulations

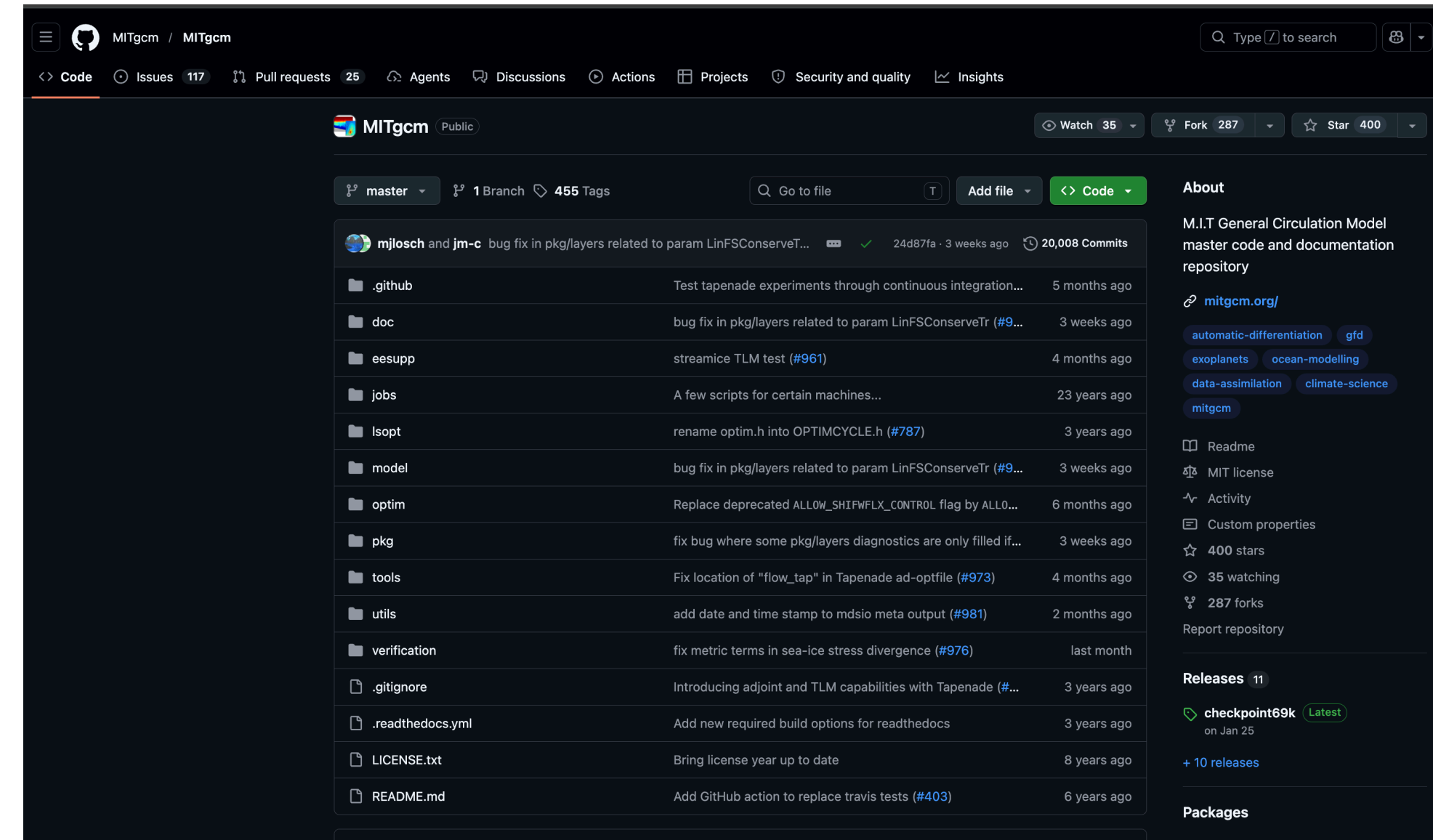
Challenges Behind Modern Day Earth Systems: Models



- Differentiability enables powerful algorithms to analyze models (inverses)
 - Differentiate w.r.t. boundary conditions => explore forcing sensitivities.
 - Differentiate w.r.t. initial conditions => produce optimal forecast.
 - Differentiate w.r.t. parameters => calibrate the model to data.
 - Substitute parameterized schemes with a surrogate neural network.

Challenges Behind Modern Day Earth Systems: Code

- Big existing codebases:
 - MITGCM: 800k LOC (Fortran)
- Developed by entire institutions (DOE, NASA, Max Planck Institute, etc)
- Experts in scientific side of simulation, less on computation front
- Sweeping changes are *hard*



Challenges Behind Modern Day Earth Systems

- Four big challenges that make running simulations hard:
 - Scale
 - Data
 - Models
 - Code
- How can we make this work?

Challenges Behind Modern Day Earth Systems

- Four big challenges that make running simulations hard:
 - Scale
 - Data
 - Models
 - Code
- How can we make this work?
 - ***By combining domain science with computational & computer science***

Challenges Behind Modern Day Earth Systems

- Four big challenges that make running simulations hard:
 - Scale
 - Data
 - Models
 - Code
- How can we make this work?
 - ***By combining domain science with computational & computer science***
 - ***By building a compiler to automate the computational domain expertise***

Outline

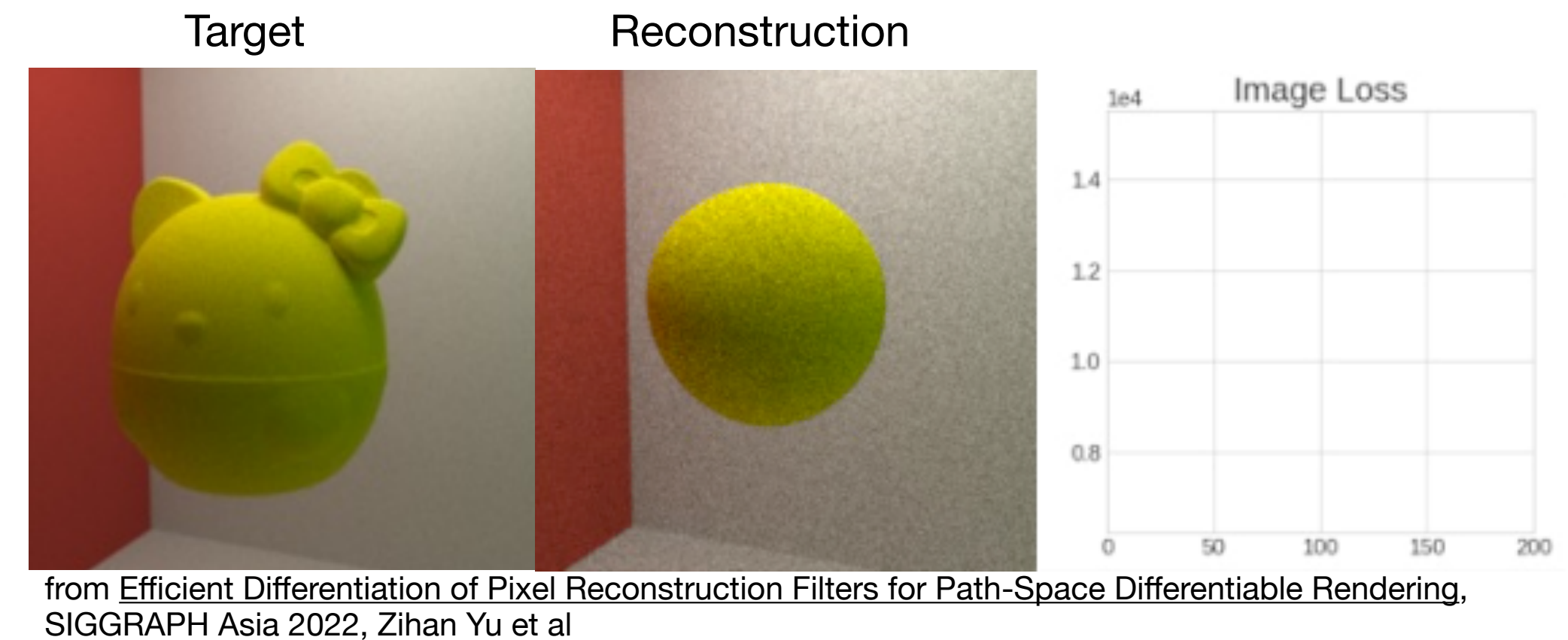
- Differentiating Existing Code
 - Memory Usage (Checkpointing)
- Performance Portability
 - Mapping to Accelerator ISAs
 - Distributed + Low Precision



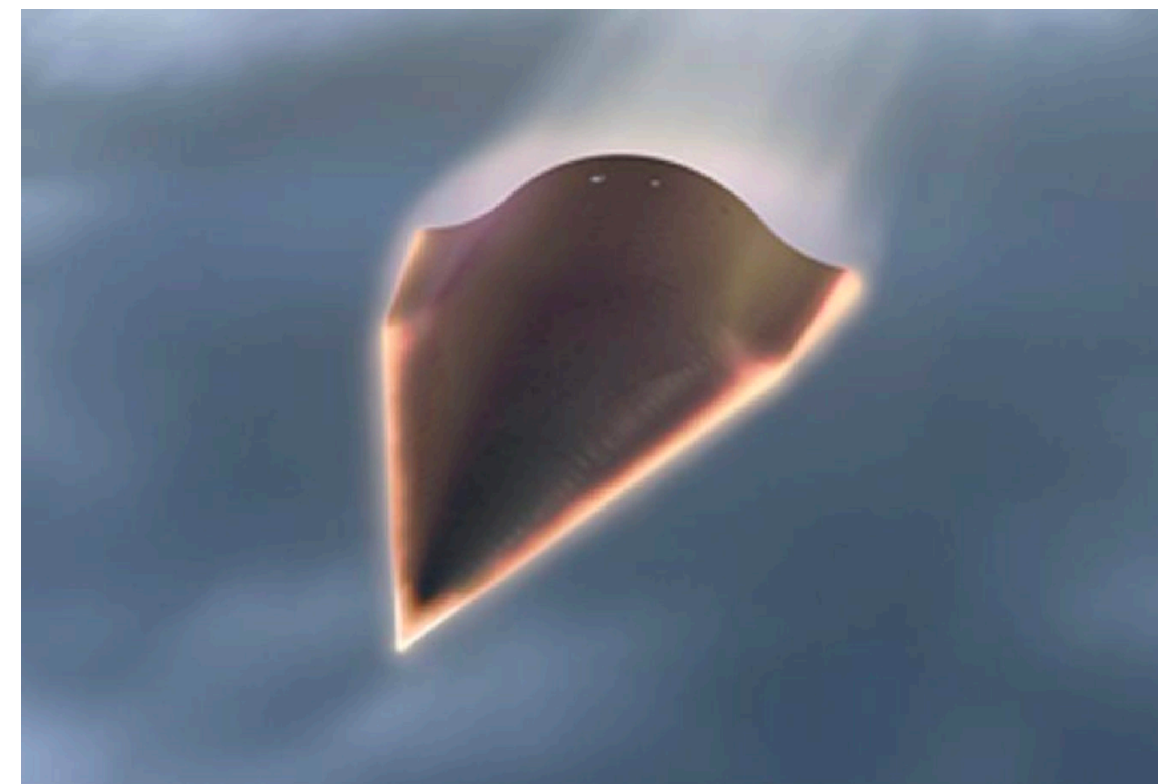
Differentiation: Connecting Science and AI

Derivatives are key to science + ML

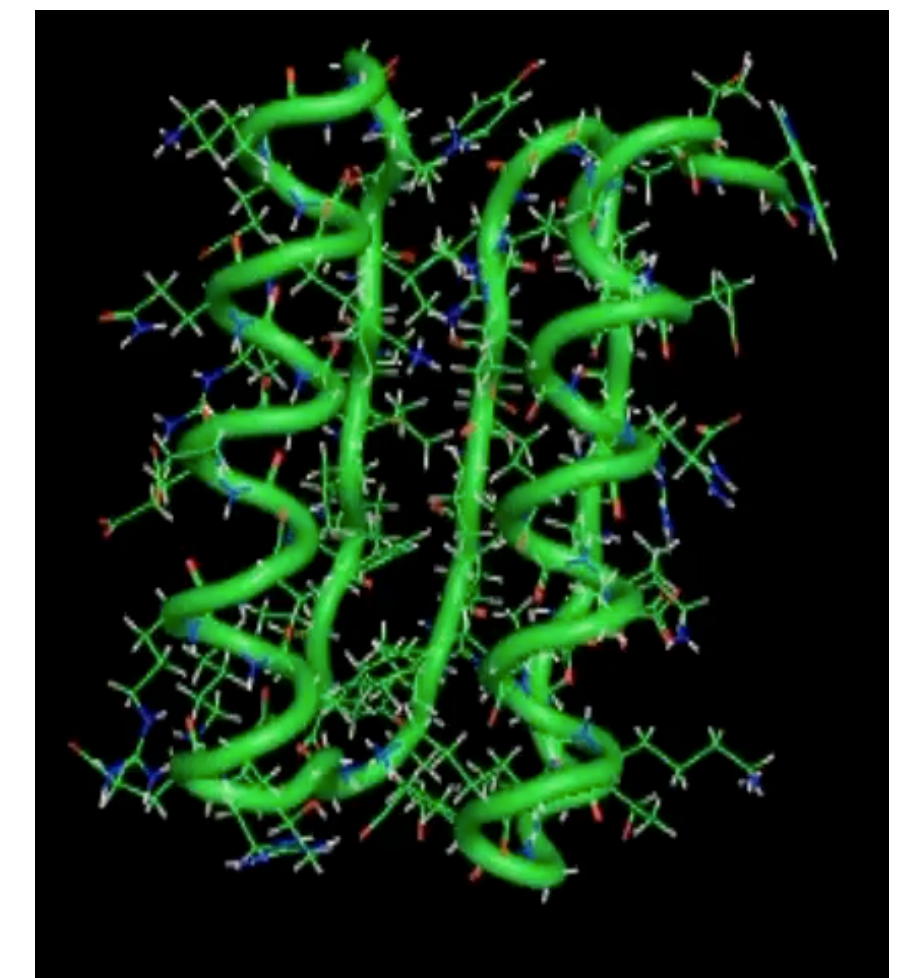
- Scientific Computing: UQ, Differential Equation, Error Analysis
- Machine Learning: Back-Propagation, Bayesian Inference



from CLIMA & NSF CSSI: Differentiable programming in Julia for Earth system modeling (DJ4Earth)



from Center for the Exascale Simulation of Materials in Extreme Environments

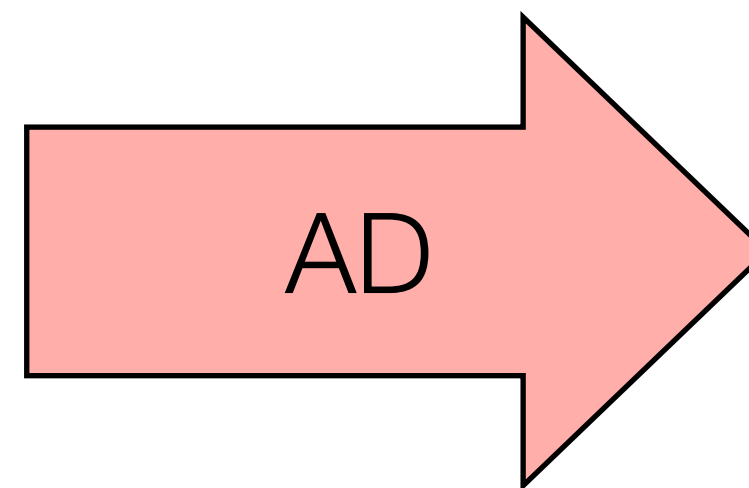


from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs

```
double relu3(double x) {  
    if (x > 0)  
        return pow(x,3)  
    else  
        return 0;  
}
```



```
double grad_relu3(double x) {  
    if (x > 0)  
        return 3 * pow(x,2)  
    else  
        return 0;  
}
```

- Unlike numerical approaches, automatic differentiation (AD) can compute the derivative of ALL inputs (or outputs) at once, without approximation error!

```
// Numeric differentiation  
// f'(x) approx [f(x+epsilon) - f(x)] / epsilon  
double grad_input[100];  
  
for (int i=0; i<100; i++) {  
    double input2[100] = input;  
    input2[i] += 0.01;  
    grad_input[i] = (f(input2) - f(input))/0.001;  
}
```

```
// Automatic differentiation  
double grad_input[100];  
  
grad_f(input, grad_input)
```

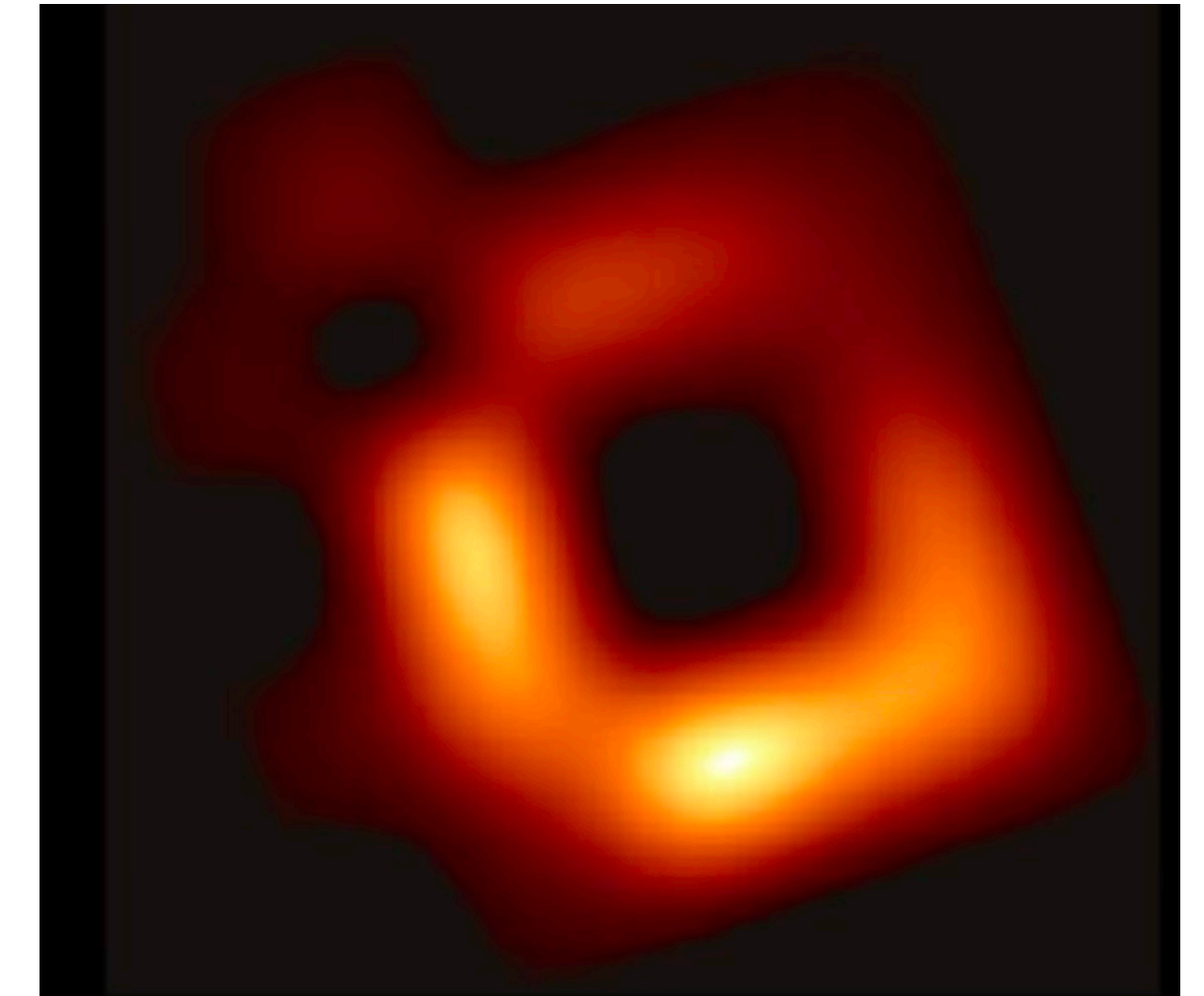

Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

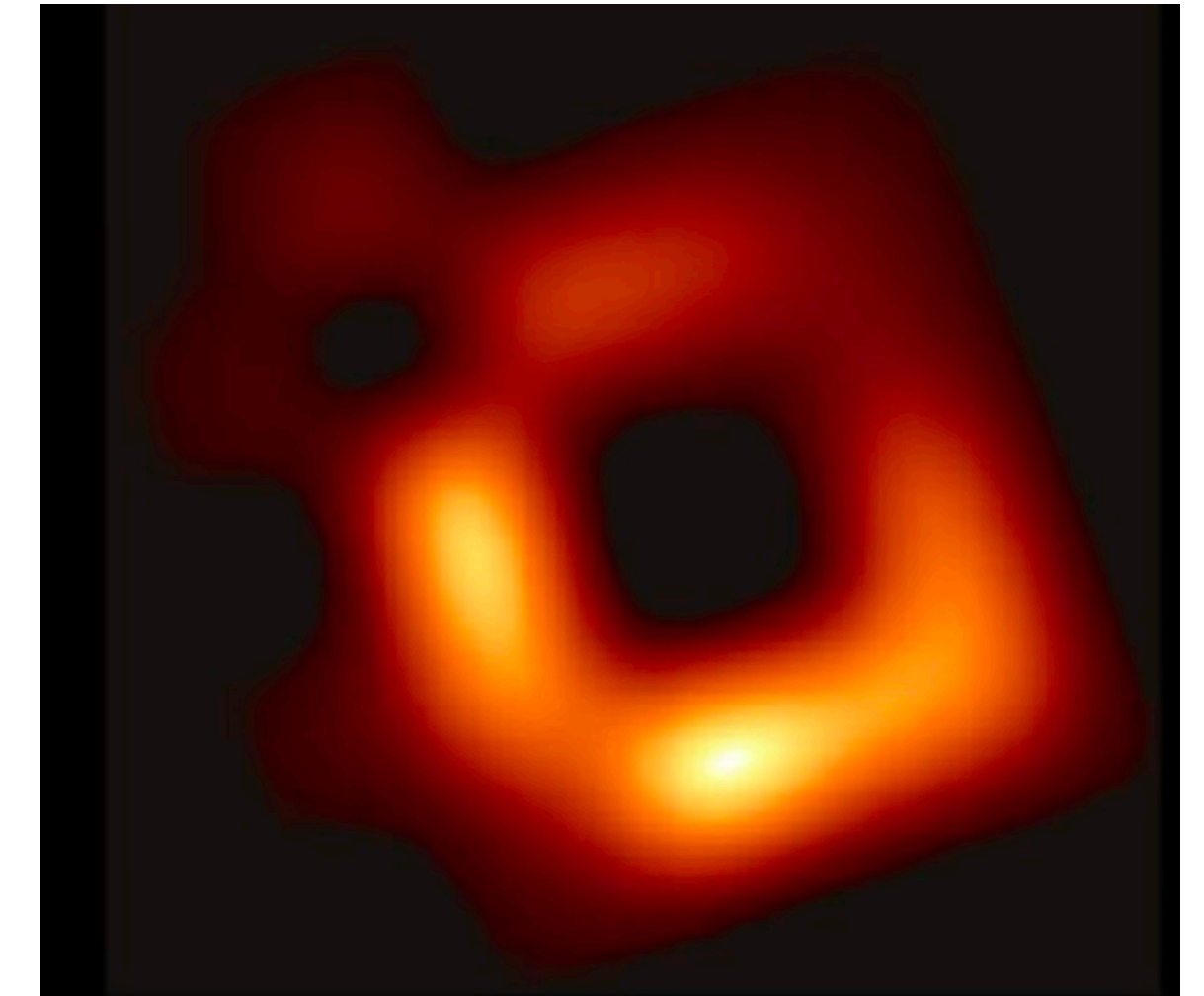
Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



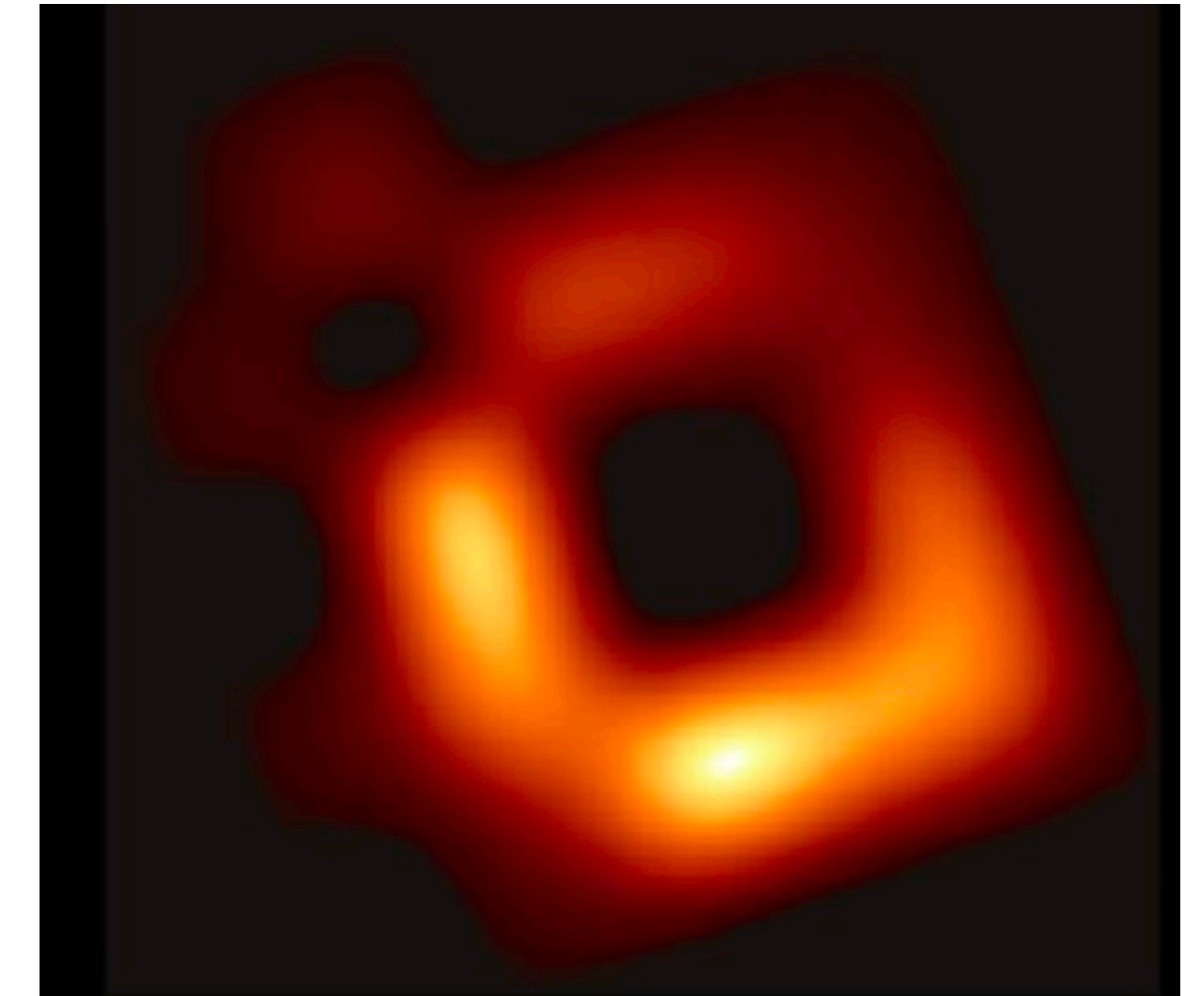
With Enzyme differentiation:
1 hour on 1 thread



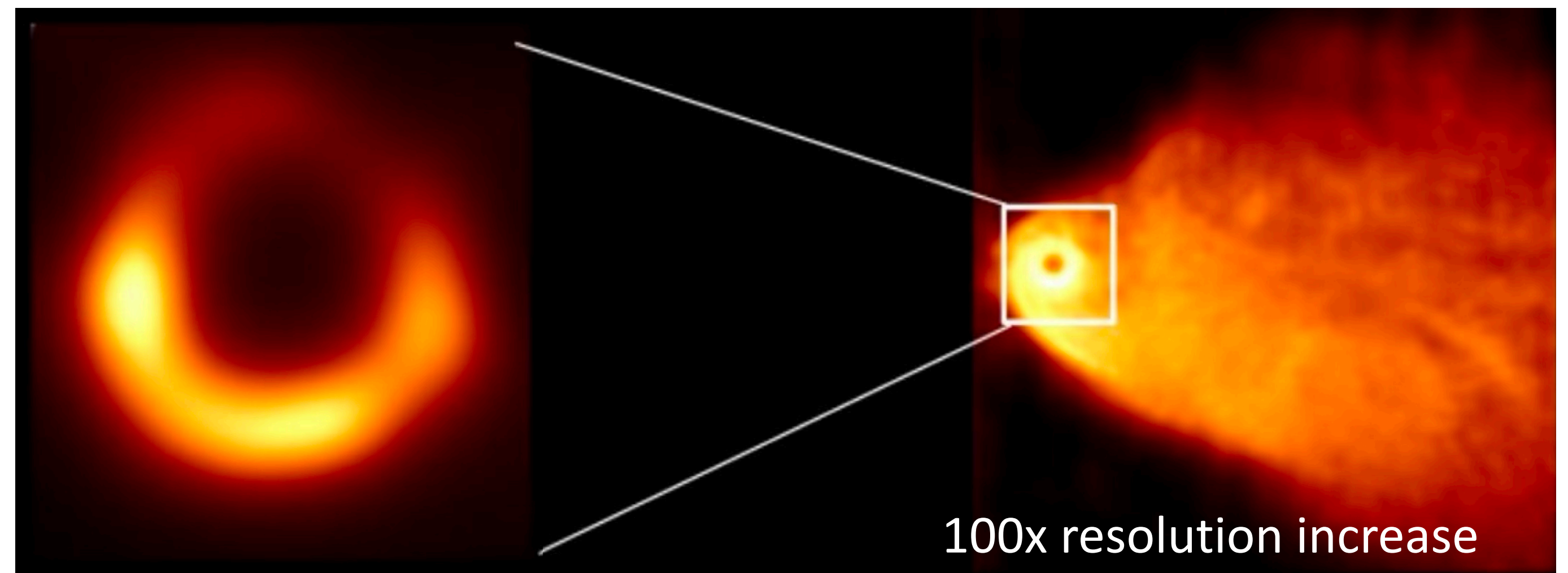
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



With Enzyme differentiation:
1 hour on 1 thread



Existing AD Approaches (1/3)

- Differentiable DSL (TensorFlow, PyTorch, DiffTaichi)
 - Provide a new language designed to be differentiated
 - Requires rewriting everything in the DSL and the DSL must support all operations in original code
 - Fast if DSL matches original code well

```
double relu3(double val) {  
    if (x > 0)  
        return pow(x,3)  
    else  
        return 0;  
}
```

Manually
Rewrite

```
import tensorflow as tf  
  
x = tf.Variable(3.14)  
  
with tf.GradientTape() as tape:  
    out = tf.cond(x > 0,  
                  lambda: tf.math.pow(x,3),  
                  lambda: 0  
    )  
print(tape.gradient(out, x).numpy())
```

Existing AD Approaches (2/3)

- Operator overloading (Adept, JAX)
 - Differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum)
 - May require writing to use non-standard utilities
 - Often dynamic: storing instructions/values to later be interpreted

```
// Rewrite to accept either
//      double or adouble
template<typename T>
T relu3(T val) {
    if (x > 0)
        return pow(x,3)
    else
        return 0;
}
```

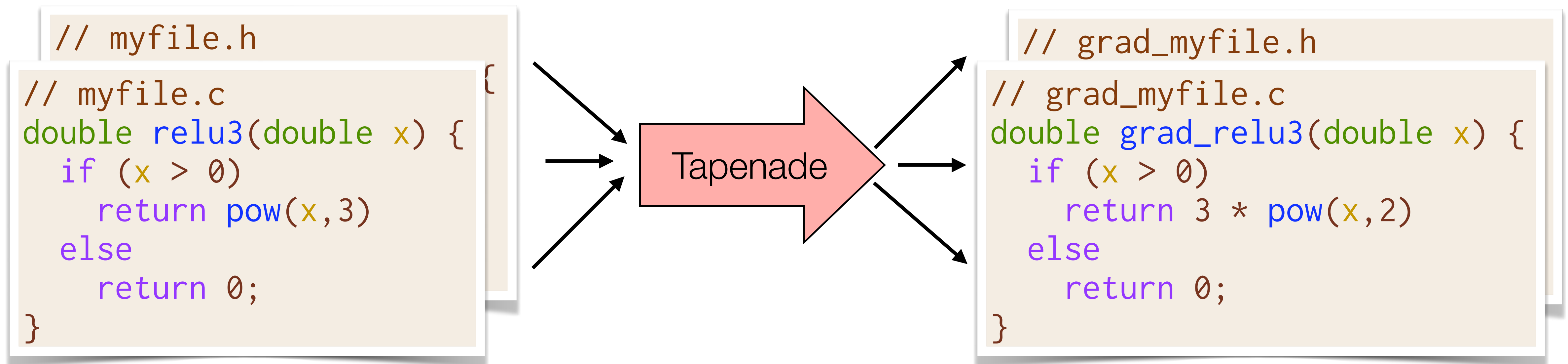
```
adept::Stack stack;
adept::adouble inp = 3.14;

// Store all instructions into stack
adept::adouble out(relu3(inp));
out.set_gradient(1.00);

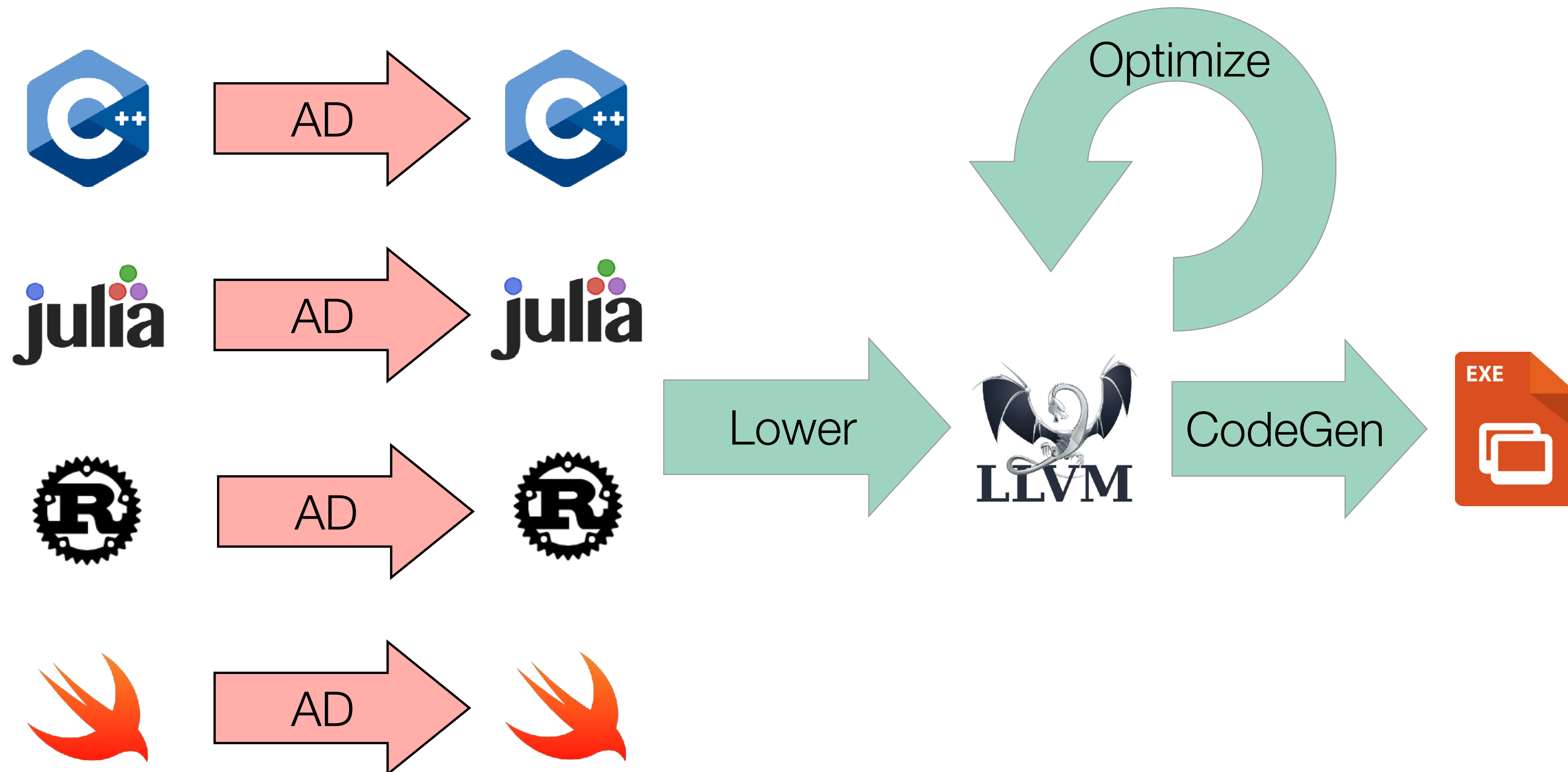
// Interpret all stack instructions
double res = inp.get_gradient(3.14);
```


Existing AD Approaches (3/3)

- Source rewriting
 - Statically analyze program to produce a new gradient function in the source language
 - Re-implement parsing and semantics of given language
 - Requires all code to be available ahead of time => hard to use with external libraries



Existing Automatic Differentiation Pipelines



Case Study: Vector Normalization

```
//Compute magnitude in O(n)
double mag(double[] x);

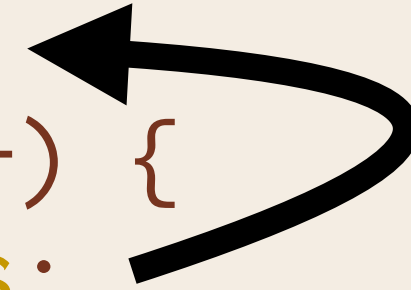
//Compute norm in O(n^2)
void norm(double[] out, double[] in) {

    for (int i=0; i<n; i++) {
        out[i] = in[i] / mag(in);
    }
}
```


Case Study: Vector Normalization

```
//Compute magnitude in O(n)
double mag(double[] x);

//Compute norm in O(n)
void norm(double[] out, double[] in) {
    double res = mag(in);
    for (int i=0; i<n; i++) {
        out[i] = in[i] / res;
    }
}
```



Optimization & Automatic Differentiation

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

Optimize

$O(n)$

```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

$O(n)$

```
d_res = 0.0  
for i=n..0 {  
  d_res += d_out[i]...  
}  
vmag(d_in, d_res)
```

Optimization & Automatic Differentiation

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

Optimize

$O(n)$

```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

$O(n)$

```
d_res = 0.0  
for i=n..0 {  
  d_res += d_out[i]...  
}  
∇mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

AD

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```


Optimization & Automatic Differentiation

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

Optimize

$O(n)$

```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

$O(n)$

```
d_res = 0.0  
for i=n..0 {  
  d_res += d_out[i]...  
}  
∇mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

AD

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

Optimize

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

Optimization & Automatic Differentiation

Differentiating after optimization can create *asymptotically faster* gradients!

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

Optimize

$O(n)$

```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

$O(n)$

```
d_res = 0.0  
for i=n..0 {  
  d_res += d_out[i]...  
}  
∇mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {  
  out[i] /= mag(in)  
}
```

AD

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

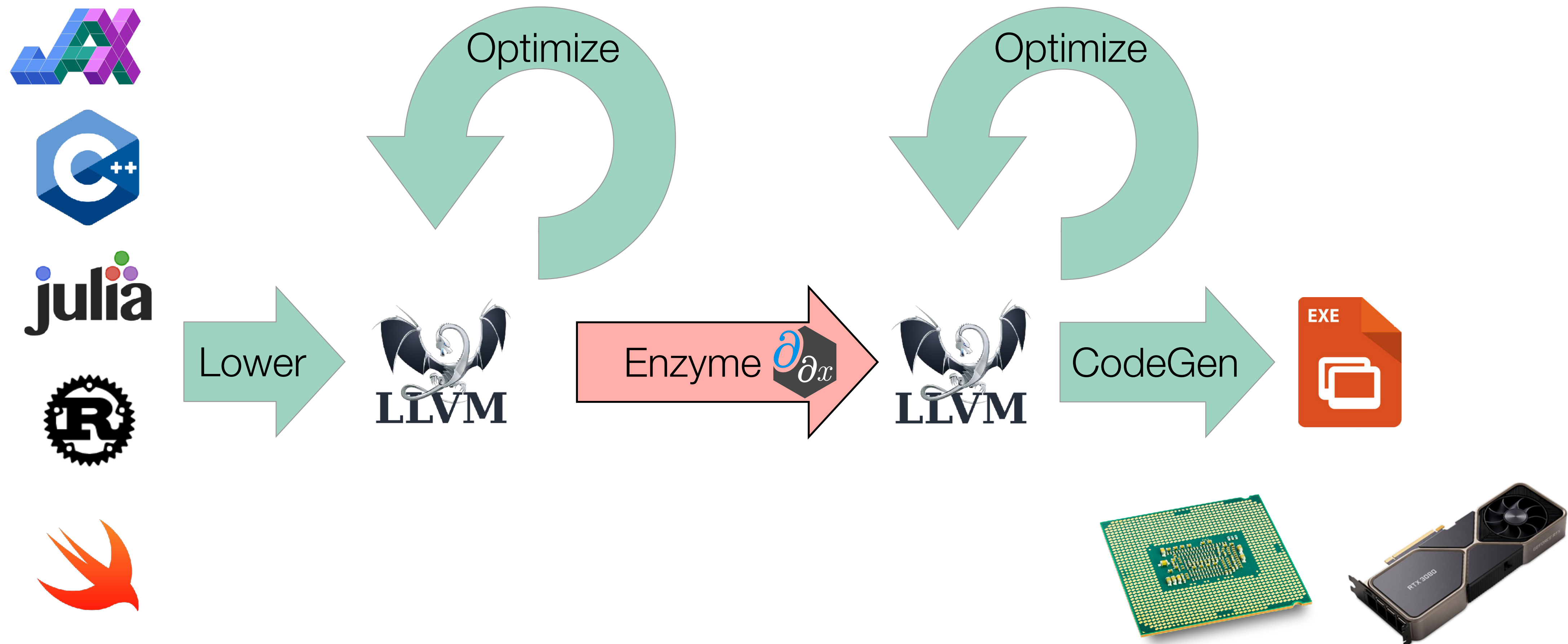
Optimize

$O(n^2)$

```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

Enzyme Approach

Performing AD at low-level lets us work on ***optimized*** code!



Case Study: ReLU3

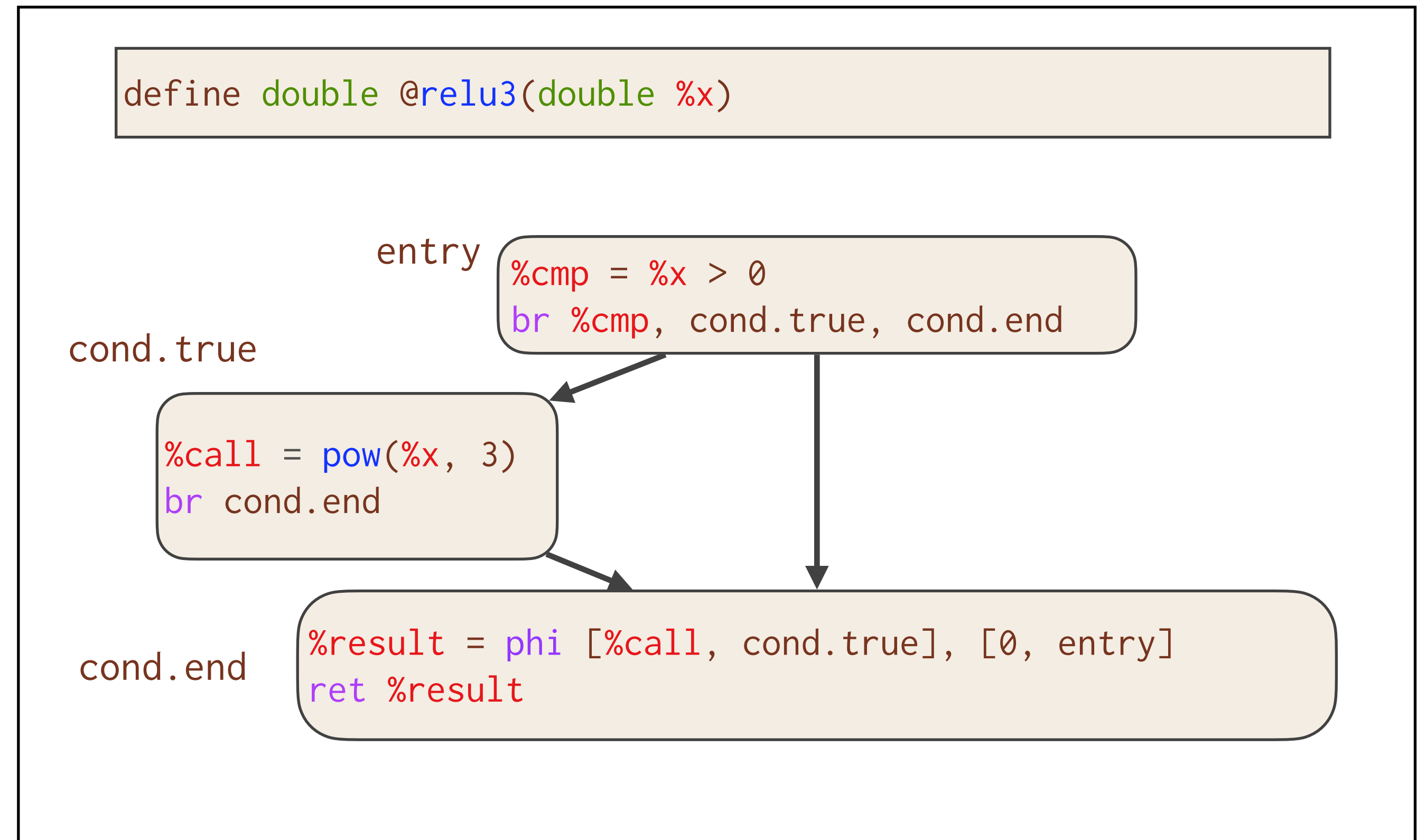
C Source

```
double relu3(double x) {  
    double result;  
    if (x > 0)  
        result = pow(x, 3);  
    else  
        result = 0;  
    return result;  
}
```

Enzyme Usage

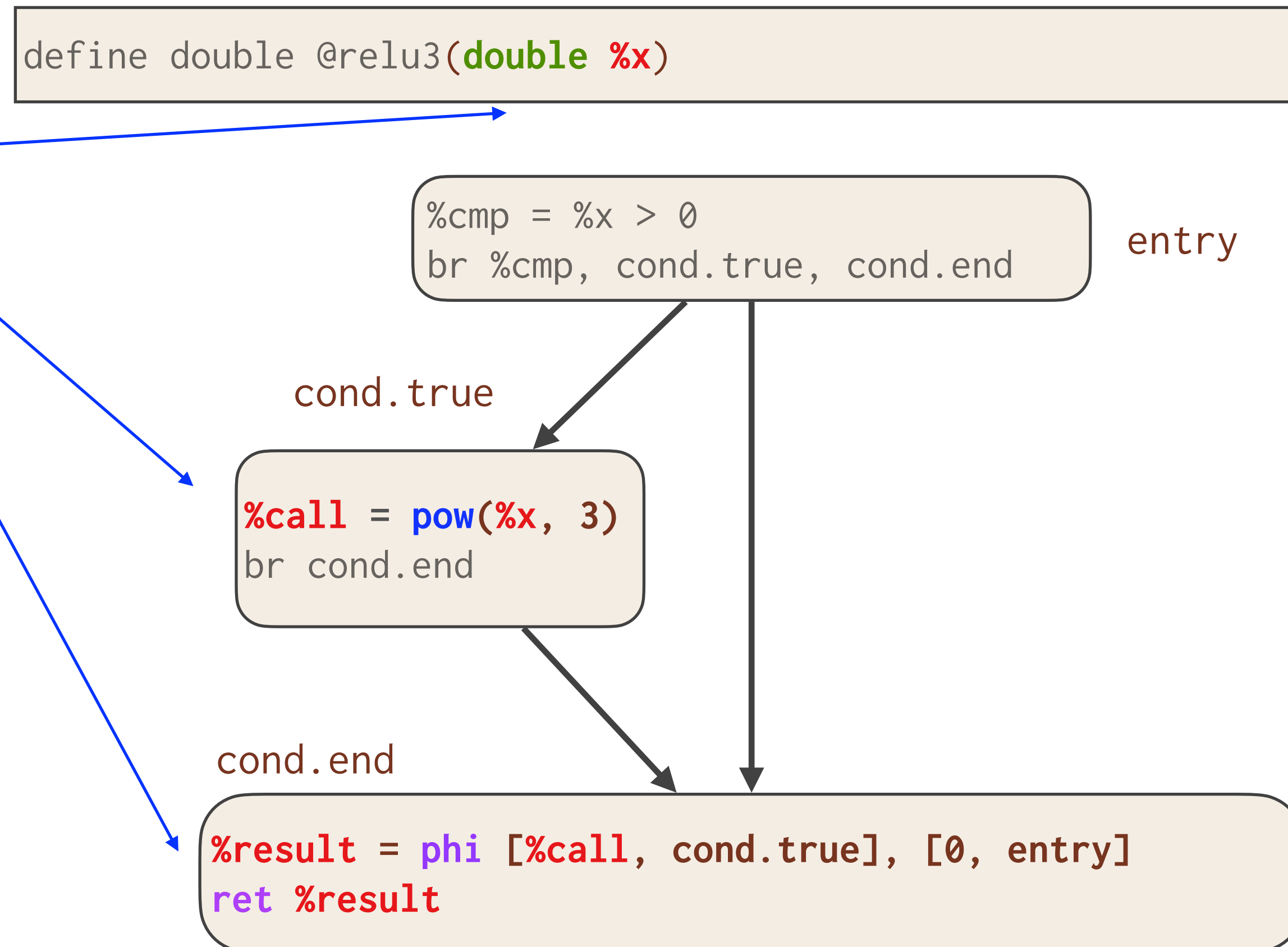
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

LLVM

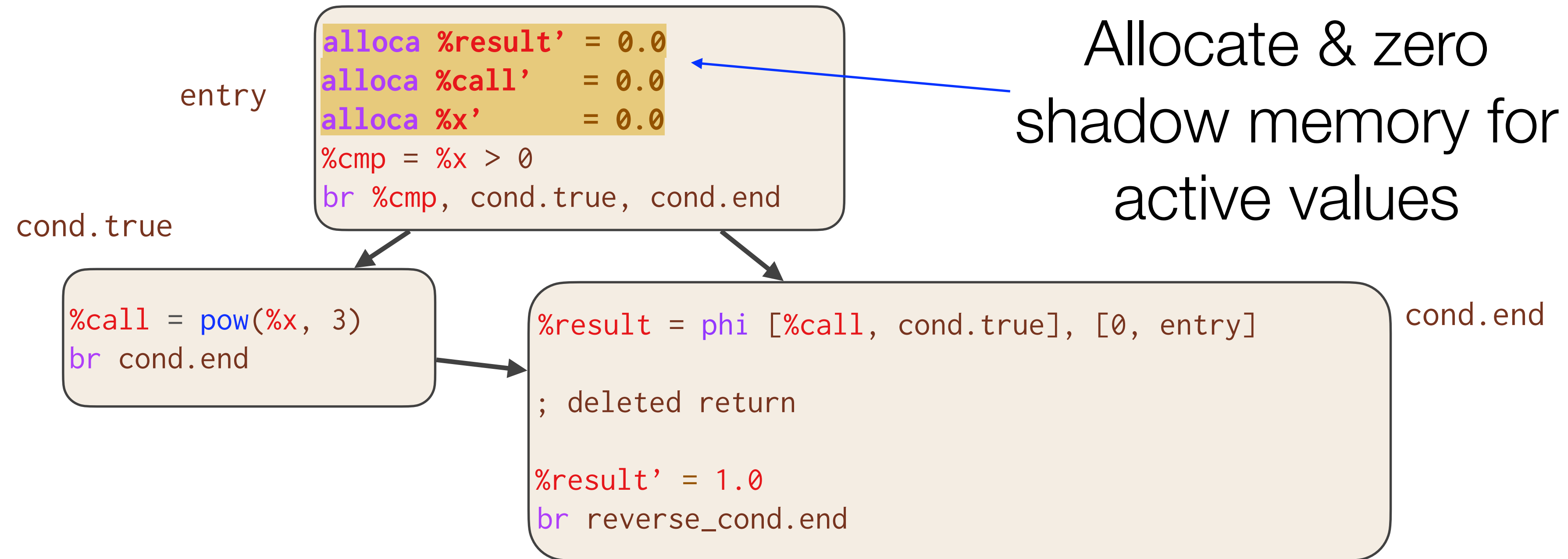


Case Study: ReLU3

Active Instructions

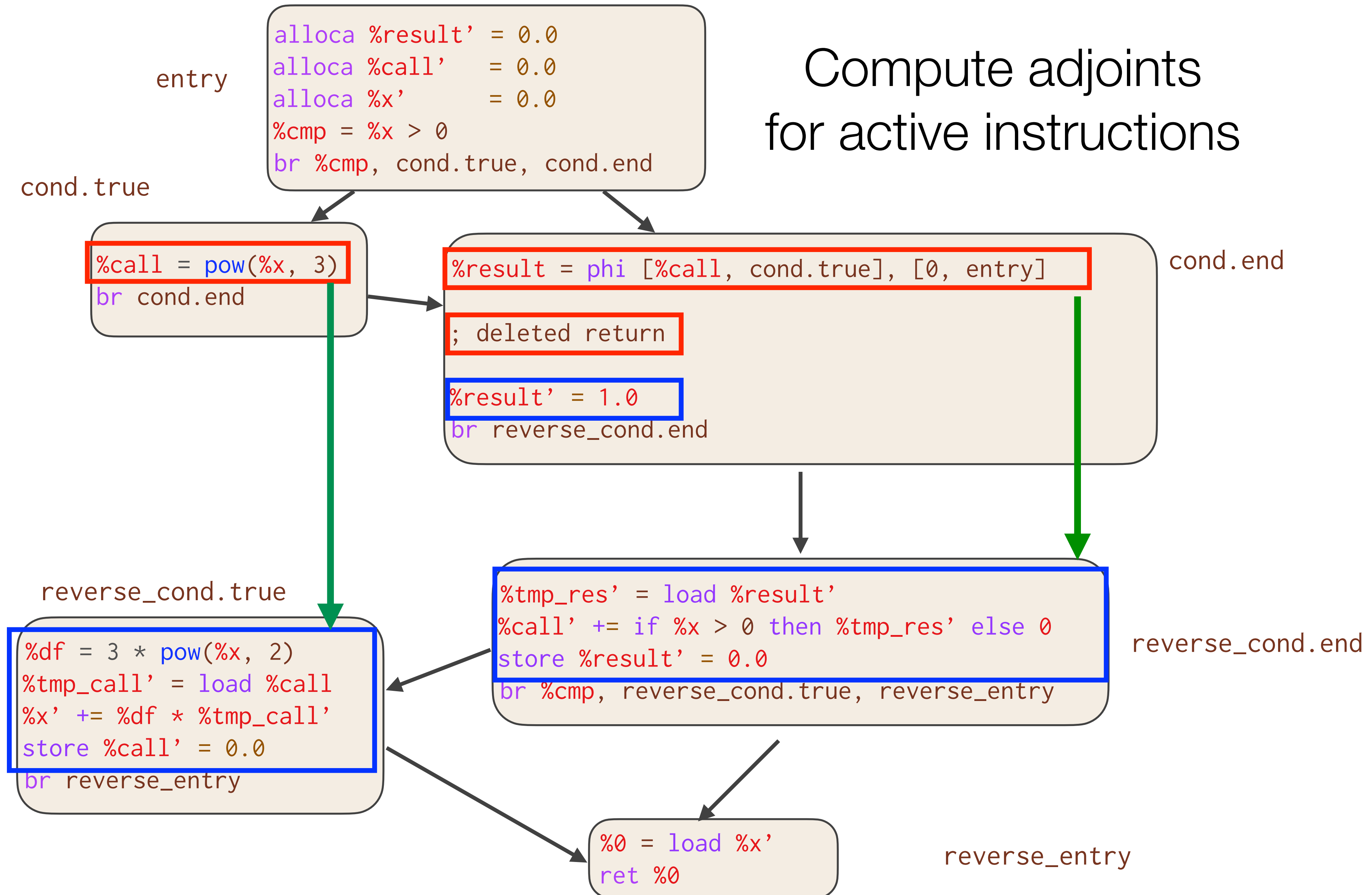


```
define double @diffe_relu3(double %x, double %differet)
```



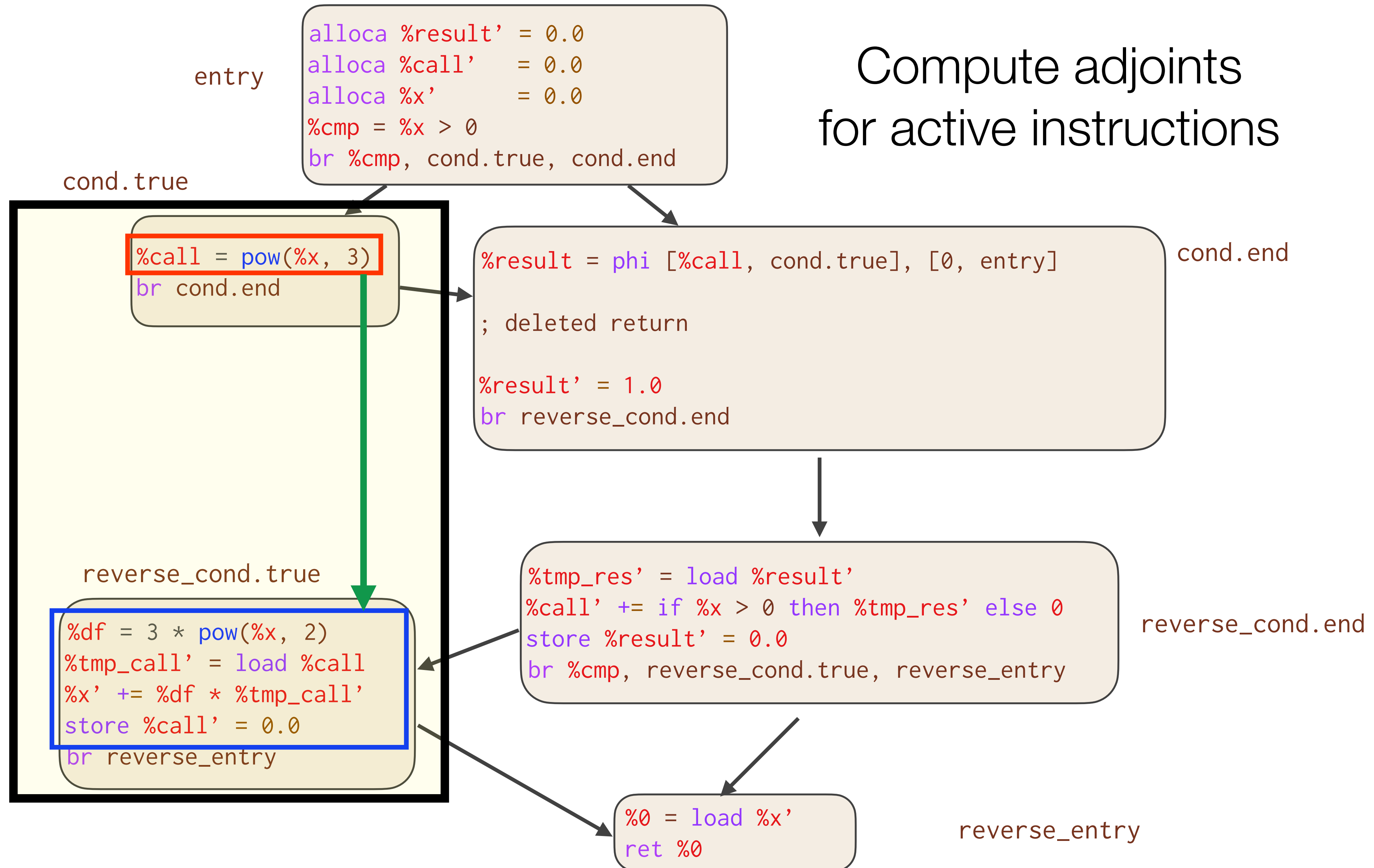
```
define double @diffe_relu3(double %x, double %differet)
```

Compute adjoints
for active instructions



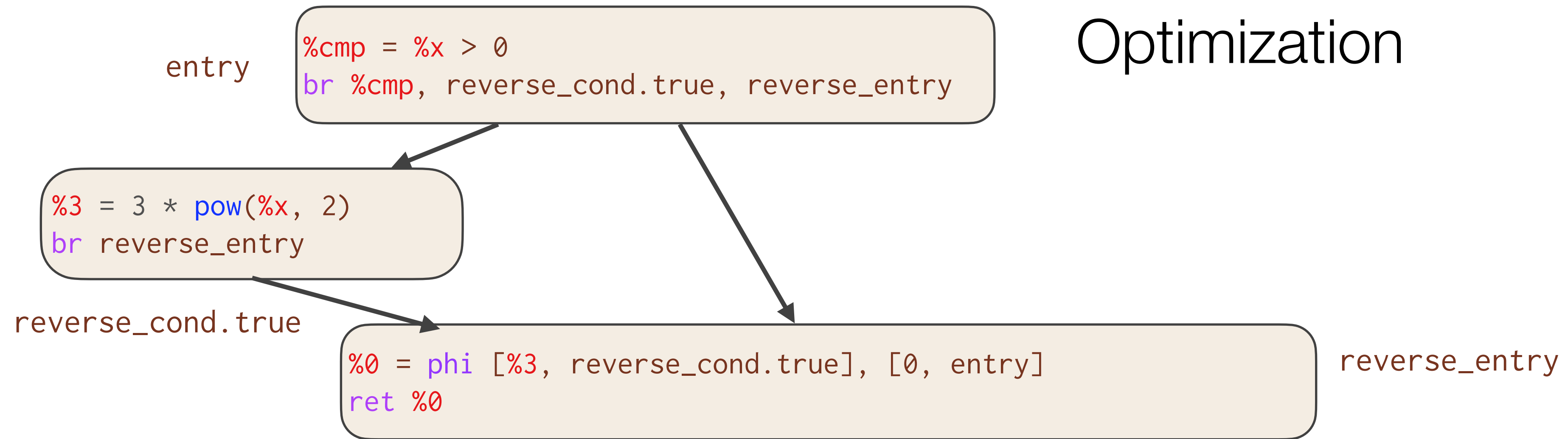

```
define double @diffe_relu3(double %x, double %differet)
```

Compute adjoints
for active instructions




```
define double @diffe_relu3(double %x)
```

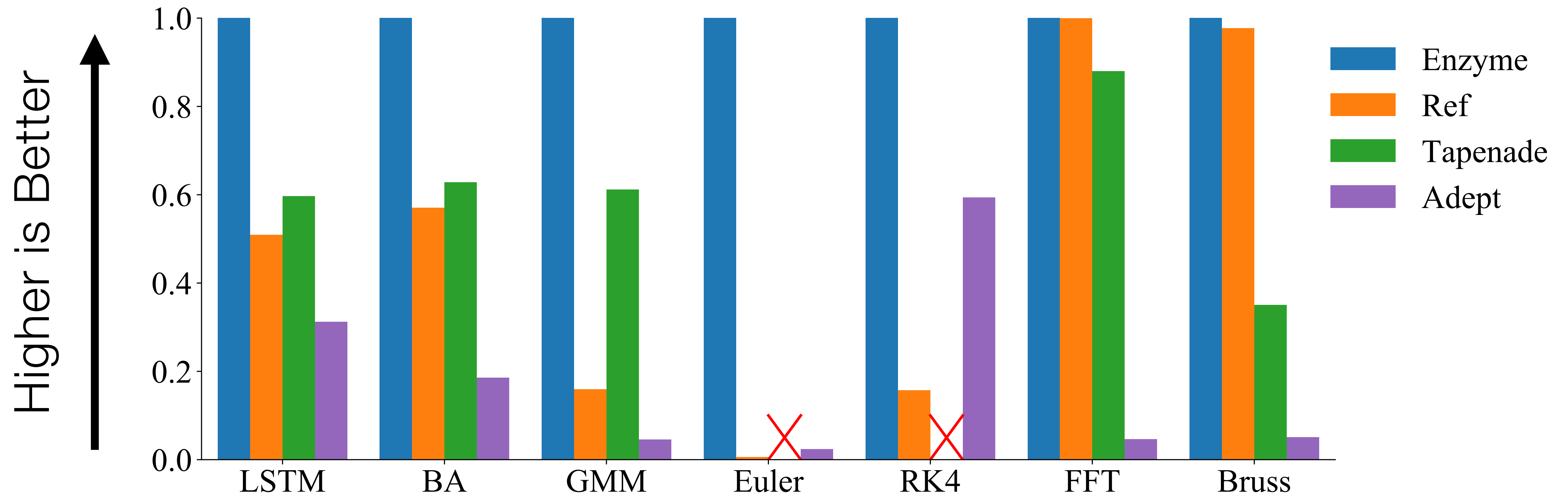
Post Optimization



Essentially the optimal hand-written gradient!

```
double diffe_relu3(double x) {  
    double result;  
    if (x > 0)  
        result = 3 * pow(x, 2);  
    else  
        result = 0;  
    return result;  
}
```

Speedup of Enzyme



Enzyme is **4.2x faster** than Reference!

Differentiation of Shallow Water Equations

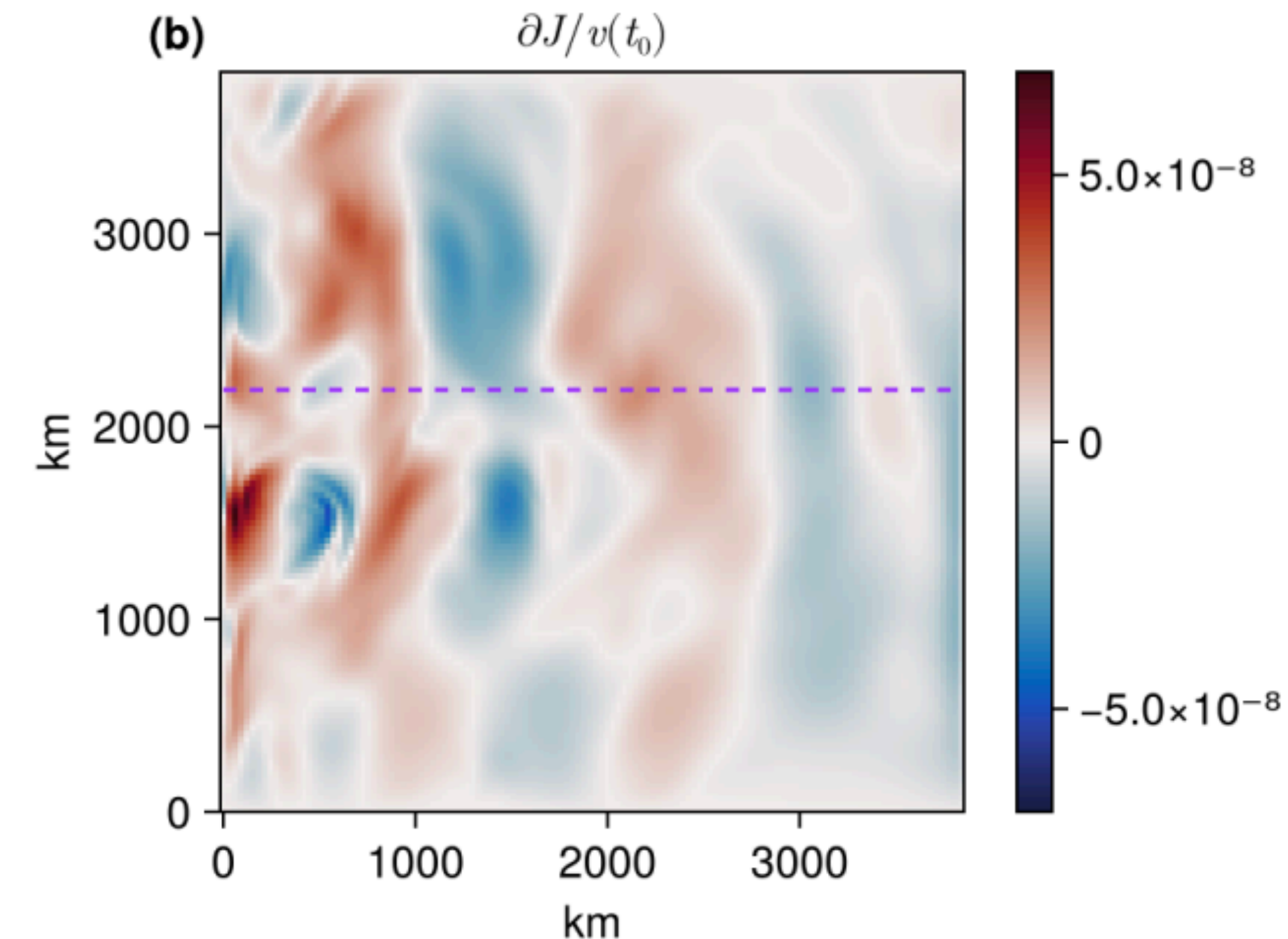
- Partial differential equations that describe fluid flow (derived from Navier Stokes)
- Start at some initial condition for some grid, integrate it forward in time
- We want to perform a sensitivity analysis: how does a given quantity of interest (QOI) at the end change if I perturb the field at the start
- J (QOI) represents kinetic energy, u/v represent momentum

$$\begin{aligned}\frac{\partial}{\partial t}u + u\frac{\partial}{\partial x}u + v\frac{\partial}{\partial y}u - fv &= -g\frac{\partial}{\partial x}\eta + M_x + F_x \\ \frac{\partial}{\partial t}v + u\frac{\partial}{\partial x}v + v\frac{\partial}{\partial y}v + fu &= -g\frac{\partial}{\partial y}\eta + M_y + F_y \\ \frac{\partial}{\partial t}\eta + \frac{\partial}{\partial x}(uh) + \frac{\partial}{\partial y}(vh) &= 0\end{aligned}$$

$$\mathcal{J}(\mathbf{u}(x, y, t_f)) = \frac{1}{N} \sum_{x,y} \{u(x, y, t_f)^2 + v(x, y, t_f)^2\},$$

Differentiation of Shallow Water Equations

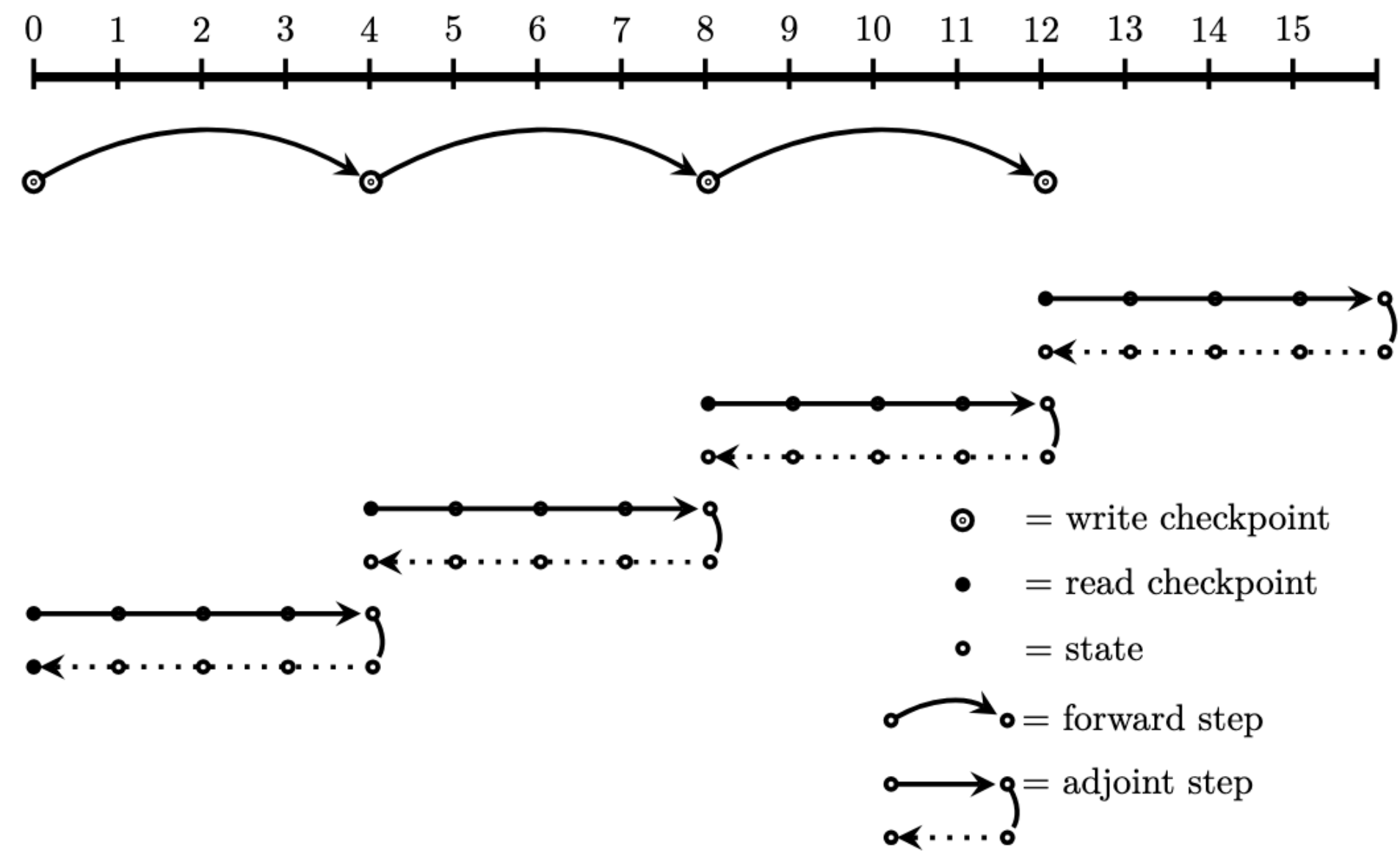
- Partial differential equations that describe fluid flow (derived from Navier Stokes)
- Start at some initial condition for some grid, integrate it forward in time
- We want to perform a sensitivity analysis: how does a given quantity of interest (QOI) at the end change if I perturb the field at the start
- J (QOI) represents kinetic energy, u/v represent momentum



$$\mathcal{J}(\mathbf{u}(x, y, t_f)) = \frac{1}{N} \sum_{x, y} \{u(x, y, t_f)^2 + v(x, y, t_f)^2\},$$

Checkpointing: Avoiding Asymptotic Memory Costs

- Checkpointing is a technique for trading off memory and compute time in the derivative

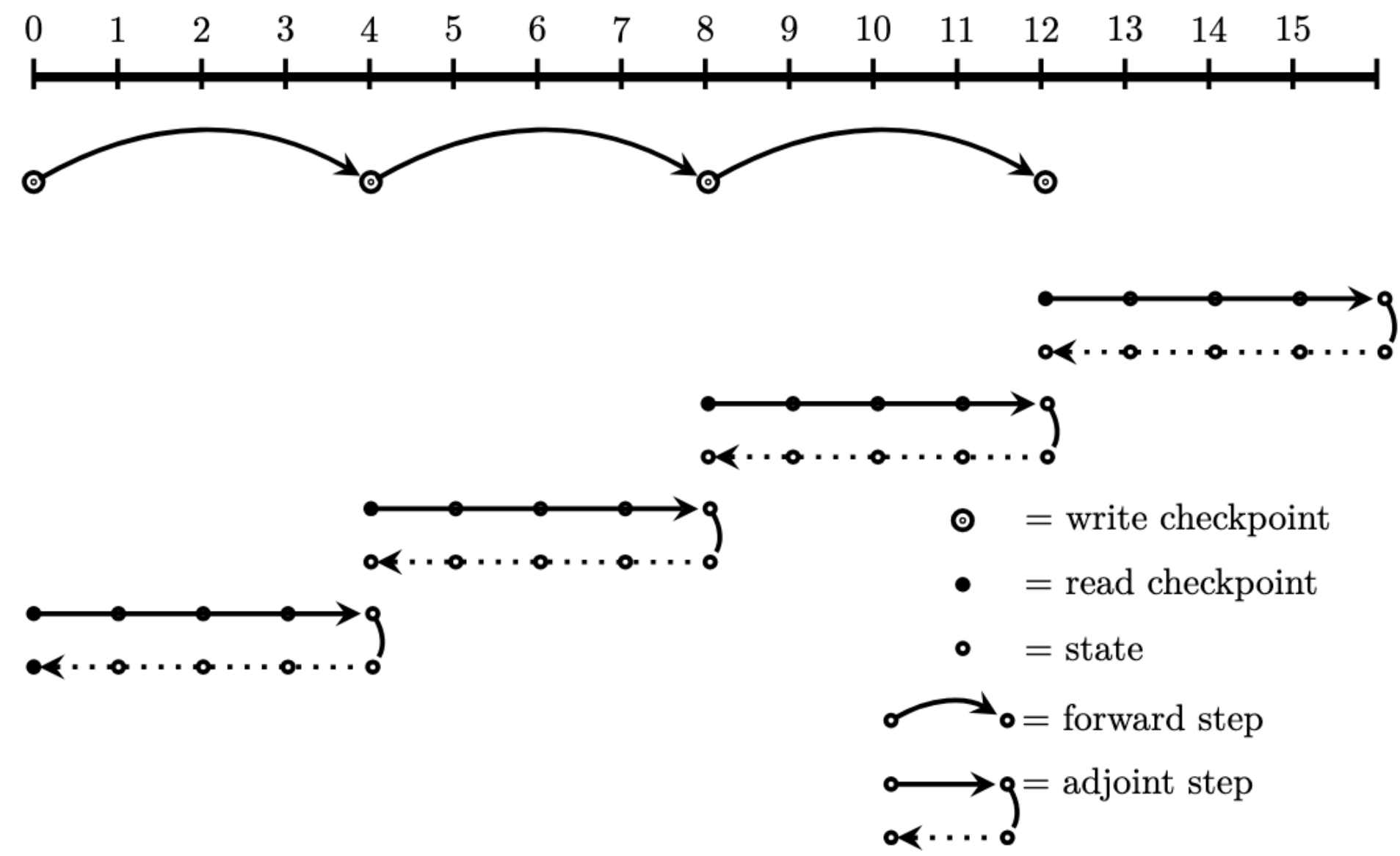


```
cache = malloc N x f32
for i = 0:N {
    x = foo(x)
    cache[i] = x
}

for i = N:0 {
    x = cache[i]
    dx = grad_foo(x, dx)
}
```

Checkpointing: Avoiding Asymptotic Memory

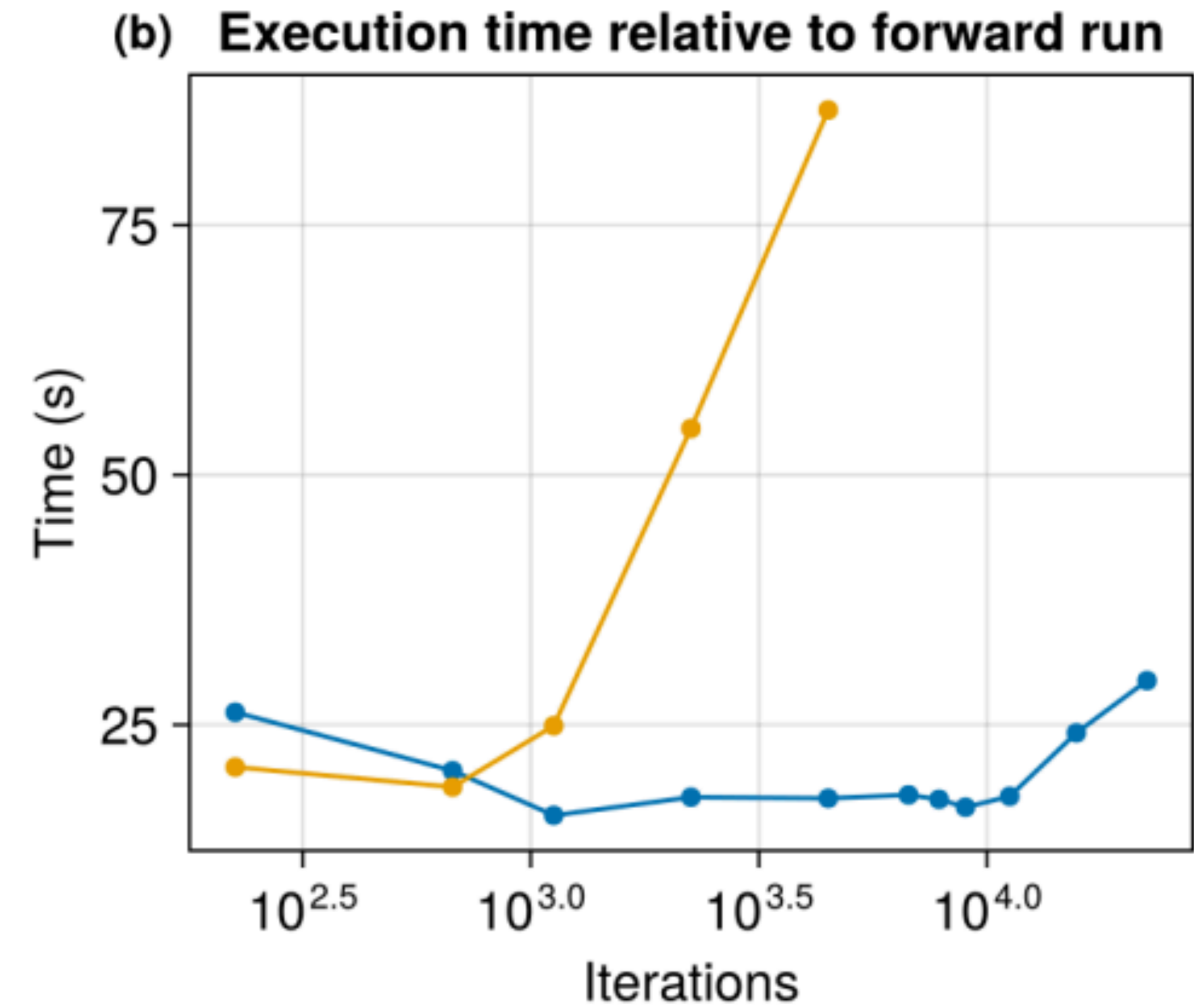
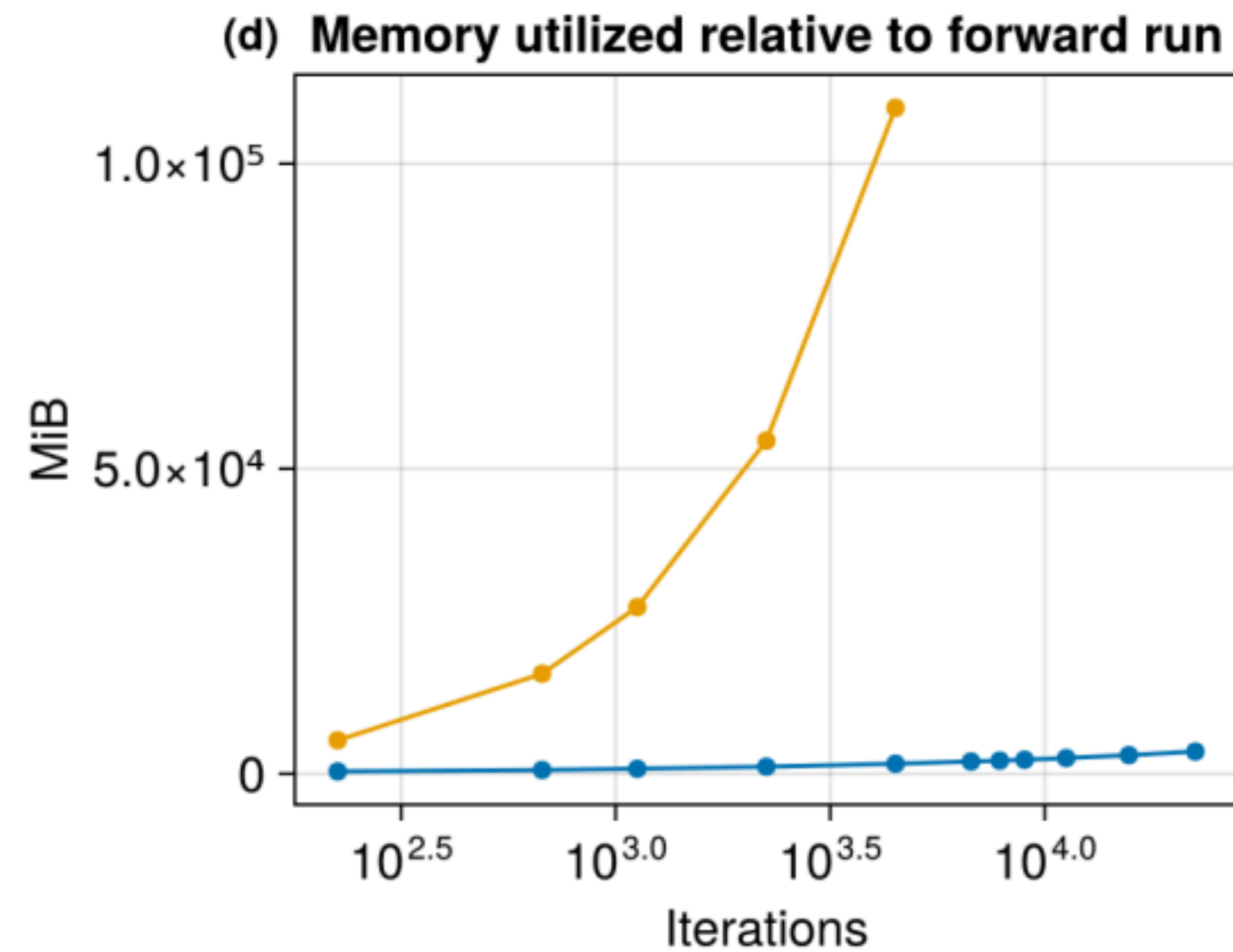
- Checkpointing is a technique for trading off memory and compute time in the derivative



```
cache = malloc M x f32
for i = 0:N/M {
    for j = 0:M {
        x = foo(x)
    }
    cache[i] = x
}

for i = N:0 {
    x = cache[i/M]
    for j in 0:i%M {
        x = foo(x)
    }
    dx = grad_foo(x, dx)
}
```

Differentiation of Shallow Water Equations



—●— With checkpointing —●— Without checkpointing

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By Katie Paul and [Krystal Hu](#)

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025



[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium. Photo by Herman/Photo Purchase Licensing Rights

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

ANTHROPIC

Claude ▾ API ▾ Solutions ▾ Research ▾ Commitments ▾ Learn ▾ News

Try Claude

Product

Claude 3.5 Haiku on AWS Tru

Ironwood: The first Google TPU for the age of inference

- When scaled to 9,216 chips per pod for a total of 42.5 Exaflops, Ironwood supports more than 24x the compute power of the world's largest supercomputer – El Capitan – which offers just 1.7 Exaflops per pod. Ironwood delivers the massive parallel processing power necessary for the most demanding AI workloads, such as super large size dense LLM or MoE models with thinking capabilities for training and inference. Each individual chip boasts peak compute of 4,614 TFLOPs. This represents a monumental leap in AI capability. Ironwood's memory and network architecture ensures that the right data is always available to support peak performance at this massive scale.

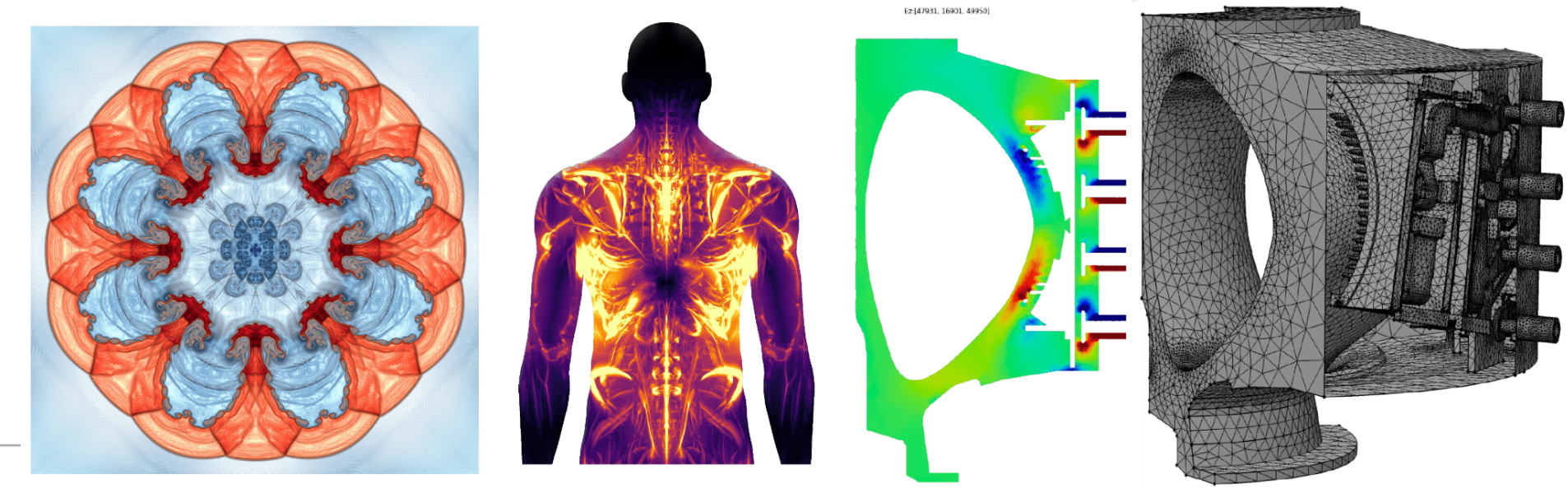
more Nvidia GPUs, an impulse all PC gamers can truly understand

News By [Andy Edser](#) published 23 July 2025

I've checked down the back of the sofa, and I'm not sure I can cover it.

[f](#) [x](#) [r](#) [p](#) [m](#) [c](#) [Comments \(2\)](#)

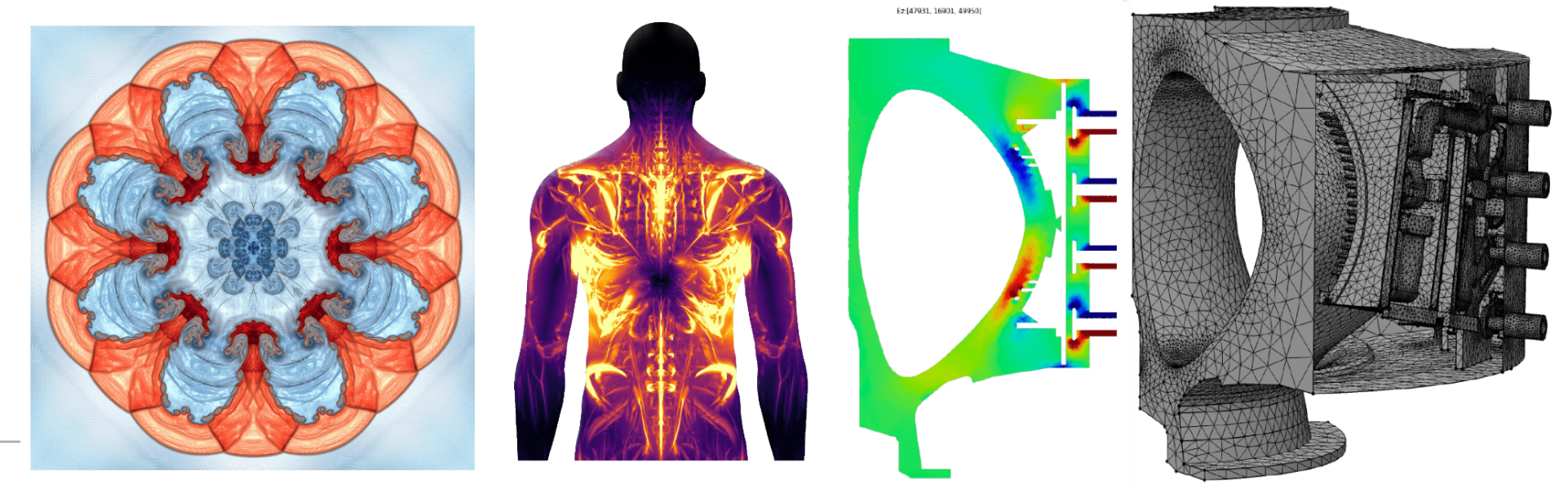
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code

```
__global__  
void AddNodeForcesFromElems_kernel( Index_t numNode,  
                                     Index_t padded_numNode,  
                                     const Int_t* nodeElemCount,  
                                     const Int_t* nodeElemStart,  
                                     const Index_t* nodeElemCornerList,  
                                     const Real_t* fx_elem,  
                                     const Real_t* fy_elem,  
                                     const Real_t* fz_elem,  
                                     Real_t* fx_node,  
                                     Real_t* fy_node,  
                                     Real_t* fz_node,  
                                     const Int_t num_threads)  
{  
    int tid=blockDim.x*blockIdx.x+threadIdx.x;  
    if (tid < num_threads)  
    {  
        Index_t g_i = tid;  
        Int_t count=nodeElemCount[g_i];  
        Int_t start=nodeElemStart[g_i];  
        Real_t fx,fy,fz;  
        fx=fy=fz=Real_t(0.0);  
  
        for (int j=0;j<count;j++)  
        {  
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here  
            fx += fx_elem[pos];  
            fy += fy_elem[pos];  
            fz += fz_elem[pos];  
        }  
  
        fx_node[g_i]=fx;  
        fy_node[g_i]=fy;  
        fz_node[g_i]=fz;  
    }  
}
```

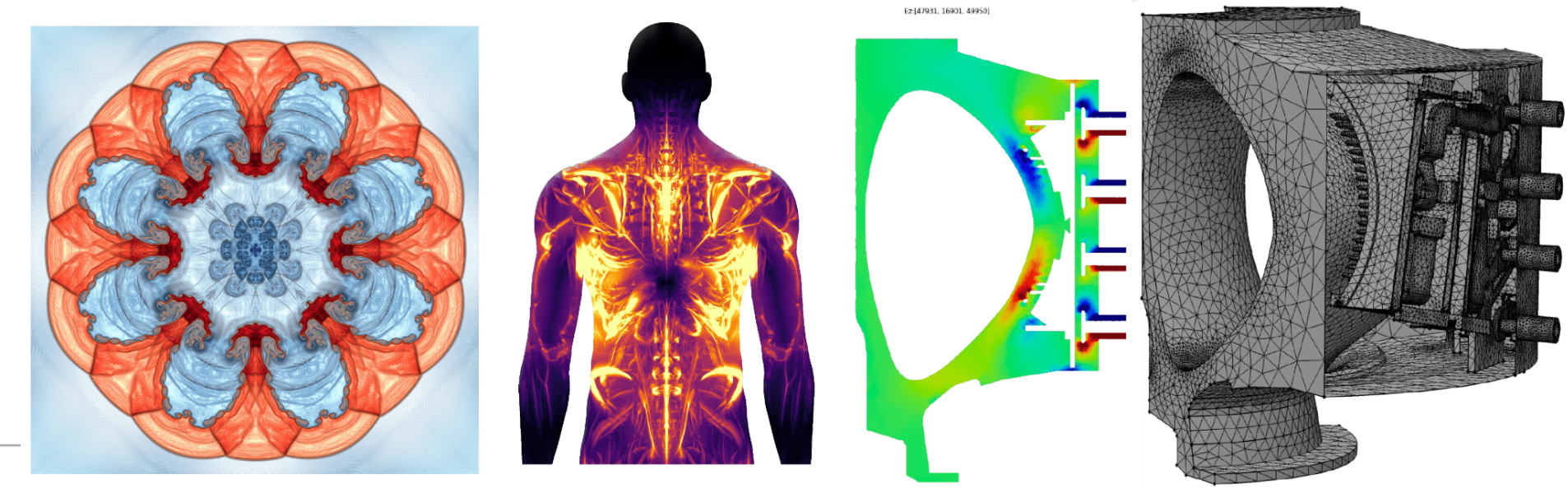
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
 - BIG (MFEM library alone is 737K LOC)

```
__global__  
void AddNodeForcesFromElems_kernel( Index_t numNode,  
                                     Index_t padded_numNode,  
                                     const Int_t* nodeElemCount,  
                                     const Int_t* nodeElemStart,  
                                     const Index_t* nodeElemCornerList,  
                                     const Real_t* fx_elem,  
                                     const Real_t* fy_elem,  
                                     const Real_t* fz_elem,  
                                     Real_t* fx_node,  
                                     Real_t* fy_node,  
                                     Real_t* fz_node,  
                                     const Int_t num_threads)  
{  
    int tid=blockDim.x*blockIdx.x+threadIdx.x;  
    if (tid < num_threads)  
    {  
        Index_t g_i = tid;  
        Int_t count=nodeElemCount[g_i];  
        Int_t start=nodeElemStart[g_i];  
        Real_t fx,fy,fz;  
        fx=fy=fz=Real_t(0.0);  
  
        for (int j=0;j<count;j++)  
        {  
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here  
            fx += fx_elem[pos];  
            fy += fy_elem[pos];  
            fz += fz_elem[pos];  
        }  
  
        fx_node[g_i]=fx;  
        fy_node[g_i]=fy;  
        fz_node[g_i]=fz;  
    }  
}
```

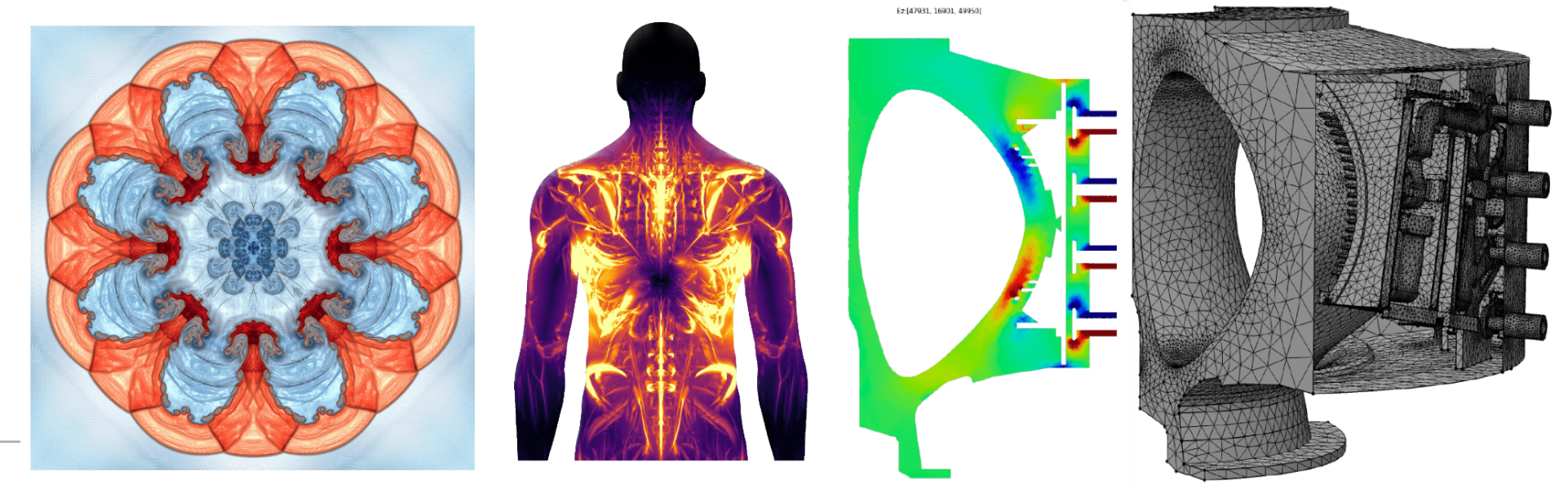

Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
 - BIG (MFEM library alone is 737K LOC)
 - Templated

```
__global__  
void AddNodeForcesFromElems_kernel( Index_t numNode,  
                                     Index_t padded_numNode,  
                                     const Int_t* nodeElemCount,  
                                     const Int_t* nodeElemStart,  
                                     const Index_t* nodeElemCornerList,  
                                     const Real_t* fx_elem,  
                                     const Real_t* fy_elem,  
                                     const Real_t* fz_elem,  
                                     Real_t* fx_node,  
                                     Real_t* fy_node,  
                                     Real_t* fz_node,  
                                     const Int_t num_threads)  
{  
    int tid=blockDim.x*blockIdx.x+threadIdx.x;  
    if (tid < num_threads)  
    {  
        Index_t g_i = tid;  
        Int_t count=nodeElemCount[g_i];  
        Int_t start=nodeElemStart[g_i];  
        Real_t fx,fy,fz;  
        fx=fy=fz=Real_t(0.0);  
  
        for (int j=0;j<count;j++)  
        {  
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here  
            fx += fx_elem[pos];  
            fy += fy_elem[pos];  
            fz += fz_elem[pos];  
        }  
  
        fx_node[g_i]=fx;  
        fy_node[g_i]=fy;  
        fz_node[g_i]=fz;  
    }  
}
```

Lingua Franca of Scientific Computing



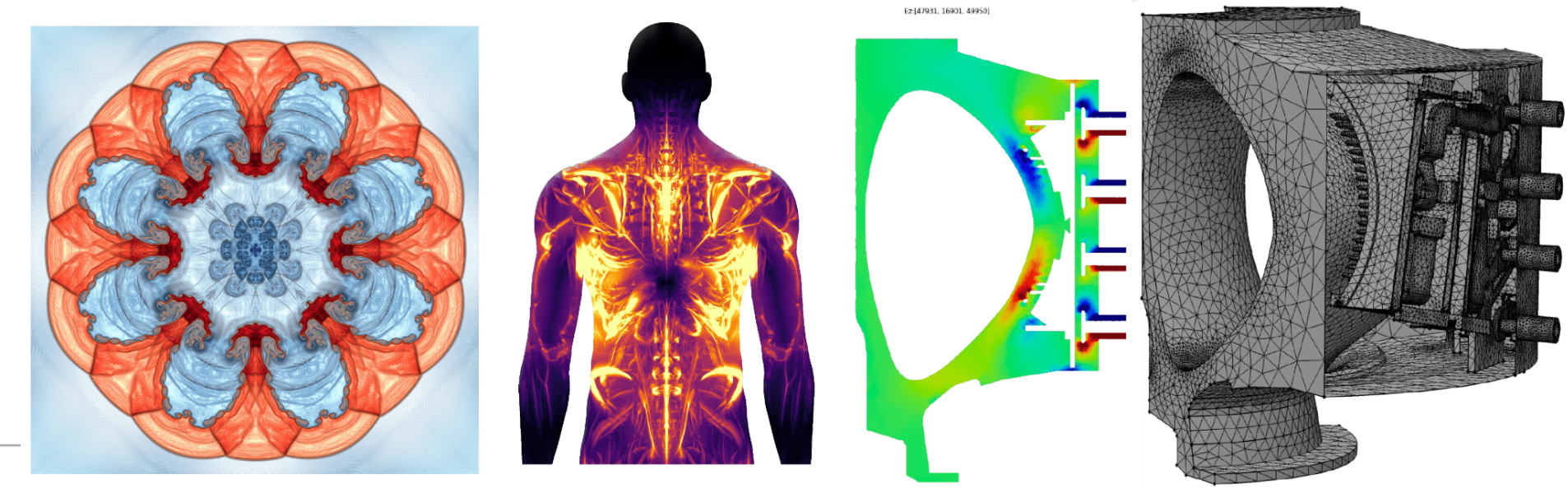
- Scientists do not write TPU* code
 - BIG (MFEM library alone is 737K LOC)
 - Templated
 - Not in Python

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                     Index_t padded_numNode,
                                     const Int_t* nodeElemCount,
                                     const Int_t* nodeElemStart,
                                     const Index_t* nodeElemCornerList,
                                     const Real_t* fx_elem,
                                     const Real_t* fy_elem,
                                     const Real_t* fz_elem,
                                     Real_t* fx_node,
                                     Real_t* fy_node,
                                     Real_t* fz_node,
                                     const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
        Index_t g_i = tid;
        Int_t count=nodeElemCount[g_i];
        Int_t start=nodeElemStart[g_i];
        Real_t fx,fy,fz;
        fx=fy=fz=Real_t(0.0);

        for (int j=0;j<count;j++)
        {
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
            fx += fx_elem[pos];
            fy += fy_elem[pos];
            fz += fz_elem[pos];
        }

        fx_node[g_i]=fx;
        fy_node[g_i]=fy;
        fz_node[g_i]=fz;
    }
}
```


Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
 - BIG (MFEM library alone is 737K LOC)
 - Templated
 - Not in Python
 - Sometimes* in CUDA

```
template <>
struct RajaCuWrap<3>
{
    template <const int BLCK = MFEM_CUDA_BLOCKS, typename DBODY>
    static void run(const int N, DBODY &&d_body,
                   const int X, const int Y, const int Z, const int G)
    {
        RajaCuWrap3D(N, d_body, X, Y, Z, G);
    }
};
```

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                     Index_t padded_numNode,
                                     const Int_t* nodeElemCount,
                                     const Int_t* nodeElemStart,
                                     const Index_t* nodeElemCornerList,
                                     const Real_t* fx_elem,
                                     const Real_t* fy_elem,
                                     const Real_t* fz_elem,
                                     Real_t* fx_node,
                                     Real_t* fy_node,
                                     Real_t* fz_node,
                                     const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
        Index_t g_i = tid;
        Int_t count=nodeElemCount[g_i];
        Int_t start=nodeElemStart[g_i];
        Real_t fx,fy,fz;
        fx=fy=fz=Real_t(0.0);

        for (int j=0;j<count;j++)
        {
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
            fx += fx_elem[pos];
            fy += fy_elem[pos];
            fz += fz_elem[pos];
        }

        fx_node[g_i]=fx;
        fy_node[g_i]=fy;
        fz_node[g_i]=fz;
    }
}
```

Looking More Deeply at Scientific Code

```
function stencil_kernel(y, x)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    if i <= length(x) - 2
        y[i] = x[i] - 2 * x[i + 1] + x[i + 2]
    end
end

function model(...)
    @cuda threads=... blocks=... stencil_kernel(y, x)
    @cuda threads=... blocks=... stencil_kernel(x, y)
end
```

> 277 such kernels

Oceananigans.jl

Fast and friendly ocean-flavored Julia software for simulating incompressible fluid dynamics in Cartesian and spherical shell domains on CPUs and GPUs. <https://clima.github.io/OceananigansDocumentation/stable>

repo status **Active** License **MIT** Ask us **anything** ColPrac **Contributor's Guide** JOSS **10.21105/joss.02018**

latest version **v0.97.7** documentation **stable release** documentation **in development**

Downloads **506/month** Total Downloads **17,790**

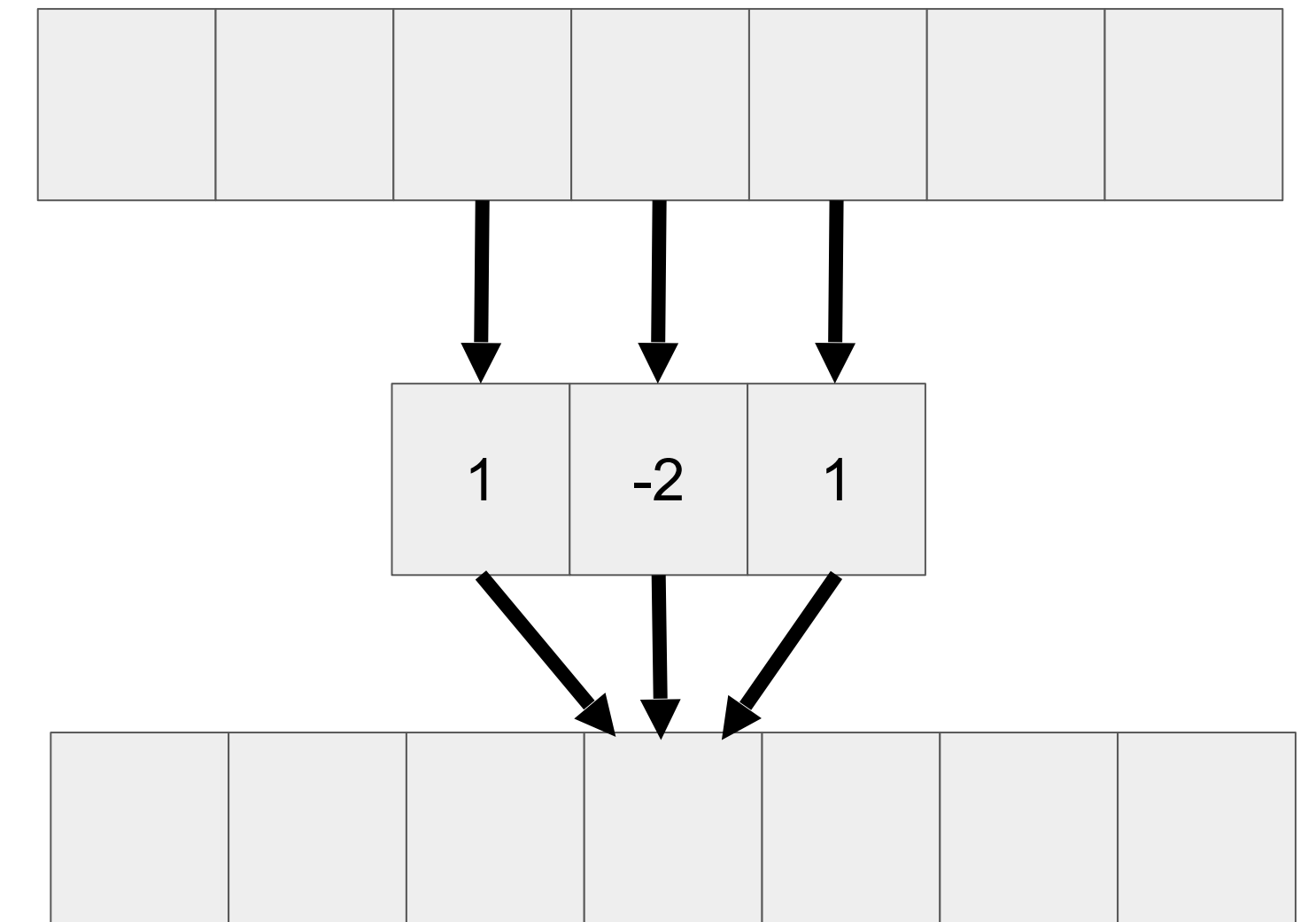
Buildkite CPU+GPU **failing** Docker **invalid**

Oceananigans is a fast, friendly, flexible software package for finite volume simulations of the nonhydrostatic and hydrostatic Boussinesq equations on CPUs and GPUs. It runs on GPUs (wow, [fast!](#)), though we believe Oceananigans makes the biggest waves with its ultra-flexible user interface that makes simple simulations easy, and complex, creative simulations possible.

Looking More Deeply at Scientific Code

```
function stencil_kernel(y, x)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    if i <= length(x) - 2
        y[i] = x[i] - 2 * x[i + 1] + x[i + 2]
    end
end

function model(...)
    @cuda threads=... blocks=... stencil_kernel(y, x)
    @cuda threads=... blocks=... stencil_kernel(x, y)
end
```



> 277 such kernels

CUDA to Accelerator IR (StableHLO)

- New framework for raising and optimizing the structure within existing kernels to stablehlo!
- 1) Compile Kernels to LLVM
- 2) Raise the underlying structure in MLIR
- 3) Multi-dimensionalize it into tensor operators
- 4) Optimize
- Compiled single-node CUDA version of code to execute on thousands of distributed TPUs and GPUs

```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i+1] + x[i+2]
  end
end

function model(...)
  @cuda threads=... blocks=... stencil_kernel(y, x)
  @cuda threads=... blocks=... stencil_kernel(x, y)
end
```

Compilation

```
define void @julia_difference_kernel_890({}* %y, {*}* %x) {
top:
  %3 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %4 = add nuw nsw i32 %3, 1
  ...
  br i1 %.not, label %common.ret, label %L31
}
```

Raising

```
func.func @kernel(%y : memref<100xf64>, %x : memref<100xf64>) {
  affine.parallel %arg1 = 0 to 100 {
    %x1 = affine.load %x[%arg1]
    %x2 = affine.load %x[%arg1 + 1]
    ...
    affine.store %sum, %y[%arg1]
  }
}
```

Multi-Dimensionalization

```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

```
res = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```


LLVM to StableHLO

LLVM/NVVM Dialect

```
llvm.call @__nv_fabsf(%arg0)
llvm.br
```

Arith + Control Flow

```
%0 = math.abs %arg0
cf.br
```

SCF (While)

```
scf.while %arg = %c0 {
  %arg < %c10
} do {
  ...
}
```

SCF (For)

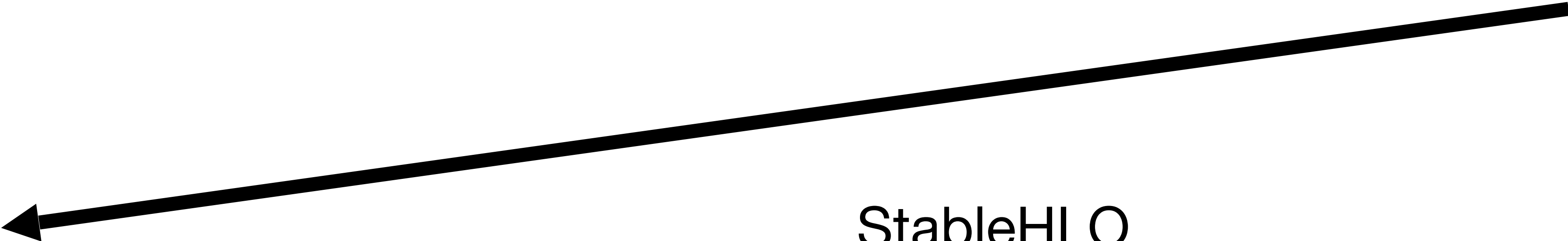
```
scf.for %arg = %c0 .. %c10 {
  ...
}
```

Affine

```
affine.for %i = 0 to 10 {
  affine.store out[%i] = ...
}
```

StableHLO

```
%x = stablehlo.slice ...
%y = stablehlo.abs %x
%z = stablehlo.dynamic_update_slice %z0[...] = %y
```



LLVM to StableHLO

LLVM/NVVM Dialect Arith + Control Flow

```
llvm.call @__nv_fabsf(%arg0)
llvm.br
```

```
%0 = math.abs %arg0
cf.br
```

High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs

William S. Moses
MIT CSAIL
Cambridge, MA, USA
wmoses@mit.edu

Ivan R. Ivanov
ivanov@mit.edu
Tokyo Tech
Japan

Jens Domke
jens.domke@riken.jp
RIKEN
Japan

Toshio Endo
endo@is.titech.ac.jp
Tokyo Tech
Japan

Johannes Doerfert
jdoerfert@lml.gov
LLNL
United States

Oleksandr Zinenko
zinenko@google.com
Google
France

Abstract
While parallelism remains the main source of performance, architectural implementations and programming models change with each new hardware generation, often leading to costly application re-engineering. Most tools for performance portability require manual and costly application porting to yet another programming model.

We propose an alternative approach that automatically translates programs written in one programming model (CUDA), into another (CPU threads) based on Polygeist/MLIR. Our approach includes a representation of parallel constructs that allows conventional compiler transformations to apply transparently and without modification and enables parallelism-specific optimizations. We evaluate our framework by transpiling and optimizing the CUDA Rodinia benchmark suite for a multi-core CPU and achieve a 38% geometric speedup over handwritten OpenMP code. Further, we show how CUDA kernels from PyTorch can efficiently run and scale on the CPU-only Supercomputer Fugaku without user intervention. Our PyTorch compatibility layer making use of transpiled CUDA PyTorch kernels outperforms the PyTorch CPU native backend by 2.7x.

1 Introduction
Despite x86 CPUs and NVIDIA GPUs remaining primary platforms for computation, customized and emerging architectures play an important role in the computing landscape. A custom version of an ARM CPU, A64FX, is even used in one of the top supercomputers Fugaku [49] where its high-bandwidth memory is expected to compete with that of GPUs. However, these architectures are often overlooked by efficiency-oriented frameworks and libraries. For example, PyTorch [44] targeting Intel's oneDNN [28] backend expectedly underperforms on ARM due to architecture differences and even Fujitsu's customized oneDNN [20] does not yield competitive performance on some kernels. Such situations call for performance portability.

Many non-library approaches for performance portability have been proposed and include language extensions (e.g., OpenCL [14], OpenACC [26]), parallel programming frameworks (e.g., Kokkos [3]), domain-specific languages (e.g., Spx, Spx, [17], Halide [47] or Tensor Comprehensions [64]). All of these approaches still require legacy applications to be ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU transpilation has been explored in the past [9, 23, 38], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transformations, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analysable representations of parallel constructs inside the compiler [39]. As representations of parallelism within a mainstream compiler have only recently begun to

Polygeist: Raising C to Polyhedral MLIR

William S. Moses*
MIT CSAIL
Cambridge, MA, USA
wmoses@mit.edu

Lorenzo Chelini*
TU Eindhoven
Eindhoven, The Netherlands
l.chelini@tue.nl

Ruizhe Zhao*
Imperial College London
London, UK
ruizhe.zhao15@imperial.ac.uk

Oleksandr Zinenko
Google Inc.
Paris, France
zinenko@google.com

Abstract—We present Polygeist, a new compilation flow that connects the MLIR compiler infrastructure to cutting edge polyhedral optimization tools. It consists of a C and C++ front-end capable of converting a broad range of existing codes into MLIR suitable for polyhedral transformation and a bi-directional conversion between MLIR and OpenScp exchange format. The Polygeist/MLIR intermediate representation featuring high-level (affine) loop constructs and n-D arrays embedded into a single static assignment (SSA) substrate enables an unprecedented combination of SSA-based and polyhedral optimizations. We illustrate this by proposing and implementing two extra transformations: statement splitting and reduction parallelization. Our evaluation demonstrates that Polygeist outperforms on average both an LLVM IR-level optimizer (Poly) and a source-to-source state-of-the-art polyhedral compiler (Pluto) when exercised on the Polybench/C benchmark suite in sequential (2.53x vs 1.41x, 2.34x) and parallel mode (9.47x vs 3.26x, 7.54x) thanks to the new representation and transformations.

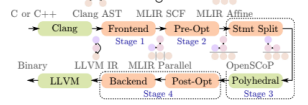


Fig. 1. The Polygeist compilation flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect (Section III-A), which is then processed by a polyhedral scheduler (Sections III-C, III-D) before post-optimization and parallelization (Section III-E). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.

1. INTRODUCTION

Improving the efficiency of computation has always been one of the prime goals of computing. Program performance can be improved significantly by reaping the benefits of parallelism, temporal and spatial locality, and other performance sources. Relevant program transformations are particularly tedious and challenging when targeting modern multicore CPUs and GPUs with deep memory hierarchies and parallelism, and are often performed automatically by optimizing compilers.

The polyhedral model enables precise analyses and a relatively easy specification of transformations (loop restructuring, automatic parallelization, etc.) that take advantage of hardware performance sources. As a result, there is growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of compute-intensive programs [1], [2], [3], and for programming accelerator architectures [4], [5], [6]. Consequently, the compiler community has focused on building tools that identify and optimize parts of the program that can be represented within the polyhedral model (commonly referred to as static-control parts, or SCOPs). Such tools tend to fall into two categories.

Compiler-based tools like Poly [7] and Graphite [8] detect and transform SCOPs in compiler intermediate representations (IRs). While this offers seamless integration with rest of the compiler, the lack of high-level structure and information hinders the tools' ability to perform analyses and transformations. This structure needs to be recovered from optimized IR, often

imperfectly or at a significant cost [9]. Moreover, common compiler optimizations such as LICM may interfere with the process [10]. Finally, low-level IRs often lack constructs for, e.g., parallelism or reductions, produced by the transformation, which makes the flow more complex.

Source-to-source compilers such as Pluto [11], PoCC [12] and PPGC [5] operate directly on C or C++ code. While this can effectively leverage the high-level information from source code, the effectiveness of such tools is often reduced by the lack of enabling optimizations such as those converting hazardous memory loads into single-assignment virtual registers. Furthermore, the transformation results must be expressed in C, which is known to be complex [13], [14] and is also missing constructs for, e.g., reduction loops or register values not backed by memory storage.

This paper proposes and evaluates the benefits of a polyhedral compilation flow, Polygeist (Figure 1), that can leverage both the high-level structure available in source code and the fine-grained control of compiler optimization provided by low-level IRs. It builds on the recent MLIR compiler infrastructure that allows the interplay of multiple abstraction levels within the same representation, during the same transformations [15]. Interchangeable MLIR abstractions, or dialects, include high-level constructs such as loops, parallel and reduction patterns; low-level representations fully covering LLVM IR [16]; and a polyhedral-inspired representation featuring loops and memory accesses annotated with affine expressions. Moreover, by combining the best of source-level and IR-level tools in an end-to-end polyhedral flow, Polygeist preserves high-level information and leverages them to perform new or improved

* Equal contribution.

Affine

```
affine.for %i = 0 to 10 {
  affine.store out[%i] = ...
}
```

StableHLO

```
%x = stablehlo.slice ...
%y = stablehlo.abs %x
%z = stablehlo.dynamic_update_slice %z0[...] = %y
```

Affine to StableHLO

- Represent *permissive, device-agnostic parallelism*
 - Legal to re-order and interchange instructions
 - One execution (lock-step), runs all of A1, then all of A2, etc
 - Lets us form efficient tensor (stablehlo) versions of kernels

```
parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){  
    %A1 = load x[%tx, %ty, %tz]  
  
    %A2 = sin(%A1)  
  
    store y[%tx, %ty, %tz] = %A2  
  
    ...  
}
```


Affine to StableHLO

- Represent *permissive, device-agnostic parallelism*
 - Legal to re-order and interchange instructions
 - One execution (lock-step), runs all of A1, then all of A2, etc
 - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]
parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){
    %A2 = sin(%A1)
    store y[%tx, %ty, %tz] = %A2
    ...
}
```


Affine to StableHLO

- Represent *permissive, device-agnostic parallelism*
 - Legal to re-order and interchange instructions
 - One execution (lock-step), runs all of A1, then all of A2, etc
 - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]
%A2 = stablehlo.sine %A1
parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){
    store y[%tx, %ty, %tz] = %A2
    ...
}
```

Affine to StableHLO

- Represent *permissive, device-agnostic parallelism*
 - Legal to re-order and interchange instructions
 - One execution (lock-step), runs all of A1, then all of A2, etc
 - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]
%A2 = stablehlo.sine %A1
%Y2 = stablehlo.dynamic_update_slice
      %Y[0:5, 0:7, 0:9], %A2
parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){
  ...
}
```

StableHLO ... to better StableHLO

- The direct vectorization of the code works, but may not be efficient.
- We will lost the convolution!
- Perform tensor-level optimizations on stablehlo to recover and optimize higher-level structures

```
%x1 = stablehlo.slice %x[1:98]  
%x2 = stablehlo.slice %x[2:99]  
%mul = stablehlo.multiply %x2, tensor<2.0>  
%add = stablehlo.add %x1, %mu  
...
```



```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>  
%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```



```
%z = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```


StableHLO ... to better StableHLO

- The direct vectorization of the code works, but may not be efficient.

- We will lose the convolution!

- ARYA VOHRA*, University of Chicago, USA
LEO SEOJUN LEE*, University of Oxford, UK
JAKUB BACHURSKI, University of Cambridge, UK
OLEKSANDR ZINENKO, Brium, France
PHITCHAYA MANGPO PHOTHILIMTHANA, OpenAI, USA
ALBERT COHEN, Google, France
WILLIAM S. MOSES, UIUC, USA
- Machine learning (ML) compilers rely on graph-level transformations to enhance the runtime performance of ML models. However, performing local transformations on individual operations can create effects far beyond the location of the rewrite. In particular, a local rewrite can change the profitability or legality of hard-to-predict downstream transformations, particularly regarding data layout, parallelization, fine-grained scheduling, and memory management. As a result, program transformations are often driven by manually-tuned compiler heuristics, which are quickly rendered obsolete by new hardware and model architectures.

Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers

ARYA VOHRA^{*}, University of Chicago, USA
LEO SEOJUN LEE^{*}, University of Oxford, UK
JAKUB BACHURSKI, University of Cambridge, UK
OLEKSANDR ZINENKO, Brium, France
PHITCHAYA MANGPO PHOTHILIMTHANA, OpenAI, USA
ALBERT COHEN, Google, France
WILLIAM S. MOSES, UIUC, USA

Machine learning (ML) compilers rely on graph-level transformations to enhance the runtime performance of ML models. However, performing local transformations on individual operations can create effects far beyond the location of the rewrite. In particular, a local rewrite can change the profitability or legality of hard-to-predict downstream transformations, particularly regarding data layout, parallelization, fine-grained scheduling, and memory management. As a result, program transformations are often driven by manually-tuned compiler heuristics, which are quickly rendered obsolete by new hardware and model architectures.

Instead of hand-written local heuristics, we propose the use of equality saturation. We replace such heuristics with a more robust *global* performance model, which accounts for downstream transformations. Equality saturation addresses the challenge of local optimizations inadvertently constraining or negating the benefits of subsequent transformations, thereby providing a solution that is inherently adaptable to newer workloads. While this approach still requires a global performance model to evaluate the profitability of transformations it holds significant promise for increased automation and adaptability.

This paper addresses challenges in applying equality saturation on real-world ML compute graphs and state-of-the-art hardware. By doing so, we present an improved method for discovering effective compositions of graph optimizations. We study different cost modeling approaches to deal with fusion and layout optimization and tackle scalability issues that arise from considering a very wide range of algebraic optimizations. We design an equality saturation pass for the XLA compiler, with an implementation in C++ and Rust. We demonstrate an average speedup of 3.45x over XLA’s optimization flow across our benchmark suite on various CPU and GPU platforms, with a maximum speedup of 56.26x for NaSNNet on CPU.

ACM Reference Format:

Arya Vohra, Leo Seojun Lee, Jakub Bachurski, Oleksandr Zinenko, Phitchaya Mangpo Phothilimthana, Albert Cohen, and William S. Moses. 2025. Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers. 1, 1 (August 2025), 28 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

*These authors contributed equally.

Authors' addresses: Arya Vohra, aryavohra@uchicago.edu, University of Chicago, USA; Leo Seojun Lee, seojun.lee@oriel.ox.ac.uk, University of Oxford, UK; Jakub Bachurski, kbachurski@gmail.com, University of Cambridge, UK; Oleksandr Zinenko, alex@brium.ai, Brium, France; Phitchaya Mangpo Phothilimthana, OpenAI, USA; Albert Cohen, albertcohen@google.com, Google, France; William S. Moses, wsmoses@illinois.edu, UTUC, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Association for Computing Machinery.
XXXX-XXXX/2025/8-ART \$15.00
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
...
```

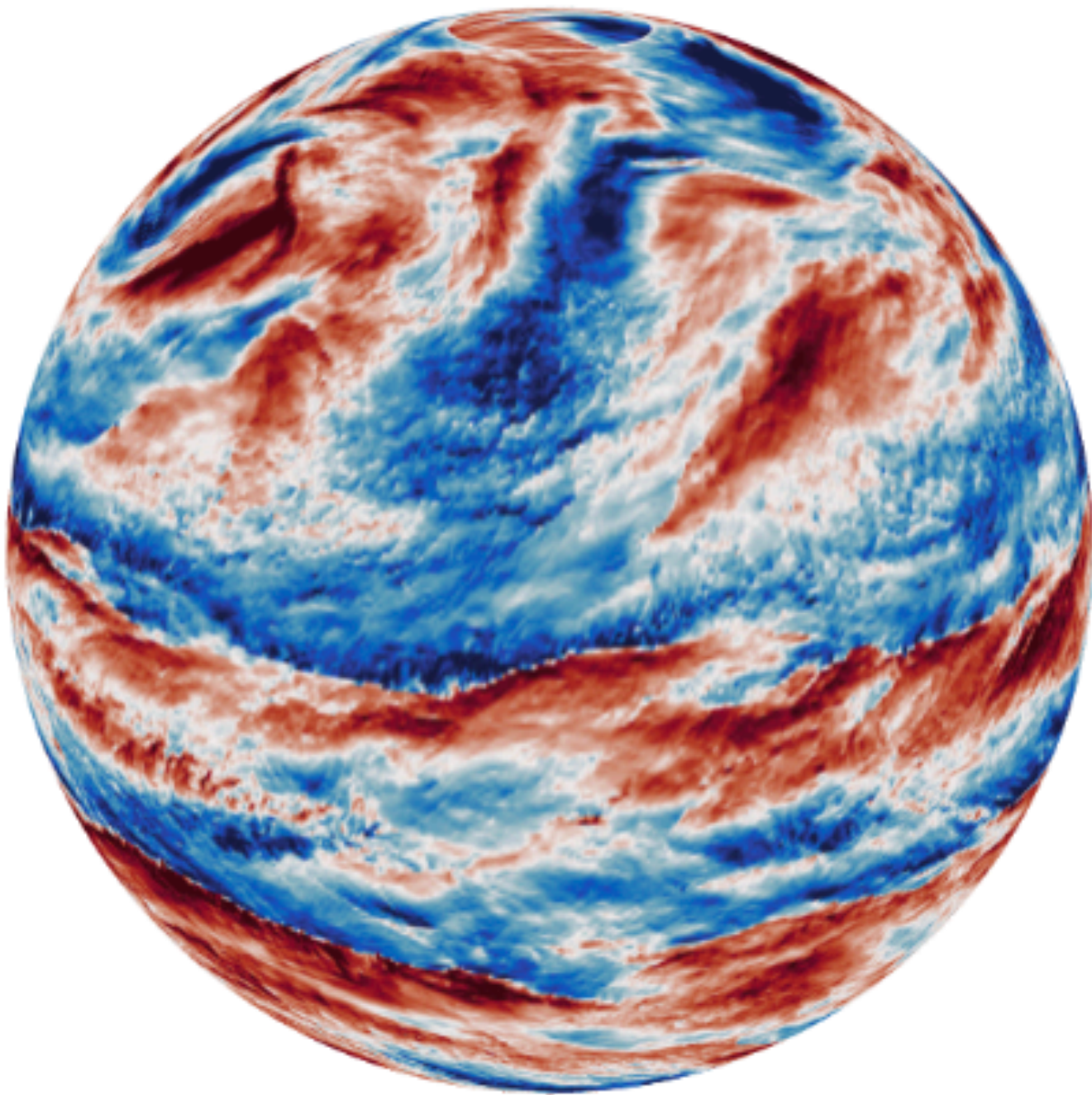
```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>
%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```

```
%z = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```

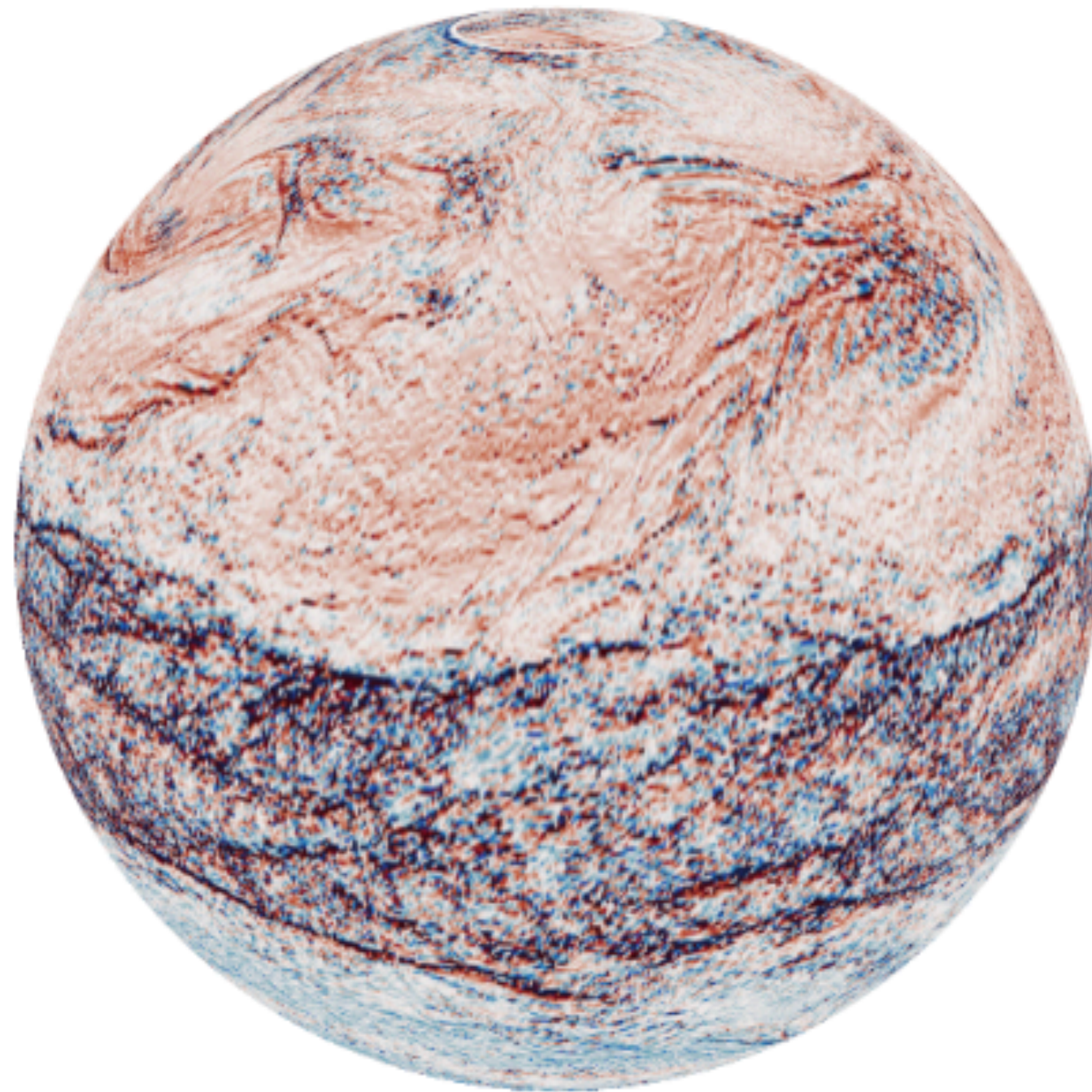
56% speedup on JaX ML workloads

CUDA to Accelerator IR (StableHLO)

Meridional momentum, p_v



Sensitivity $\partial J / \partial(\rho v_0)$
 $J = V^{-1} \int (\rho \theta)^2 dV$



```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i+1] + x[i+2]
  end
end

function model(...)
  @cuda threads=... blocks=... stencil_kernel(y, x)
  @cuda threads=... blocks=... stencil_kernel(x, y)
end
```

Compilation

```
define void @julia_difference_kernel_890({}* %y, { }* %x) {
top:
  %3 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %4 = add nuw nsw i32 %3, 1
  ...
  br i1 %.not, label %common.ret, label %L31
}
```

Raising

```
func.func @kernel(%y : memref<100xf64>, %x : memref<100xf64>) {
  affine.parallel %arg1 = 0 to 100 {
    %x1 = affine.load %x[%arg1]
    %x2 = affine.load %x[%arg1 + 1]
    ...
    affine.store %sum, %y[%arg1]
  }
}
```

Multi-Dimensionalization

```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

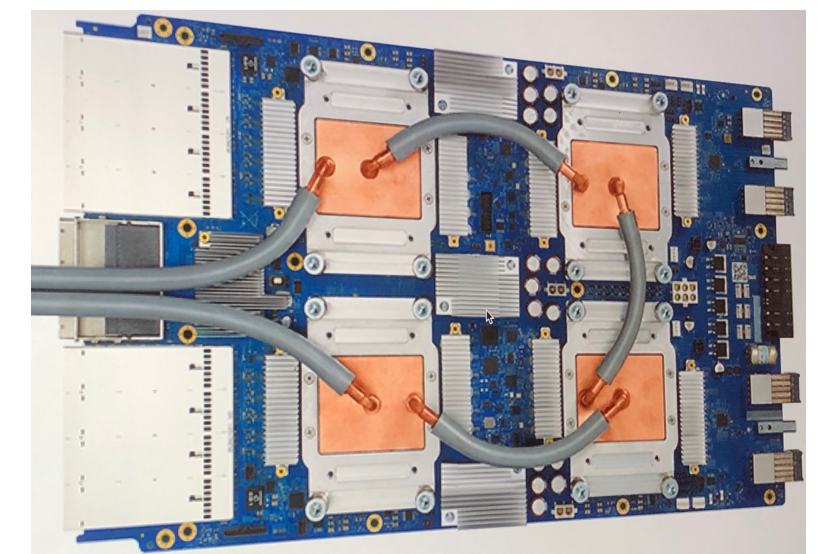
```
res = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```


Performance Results

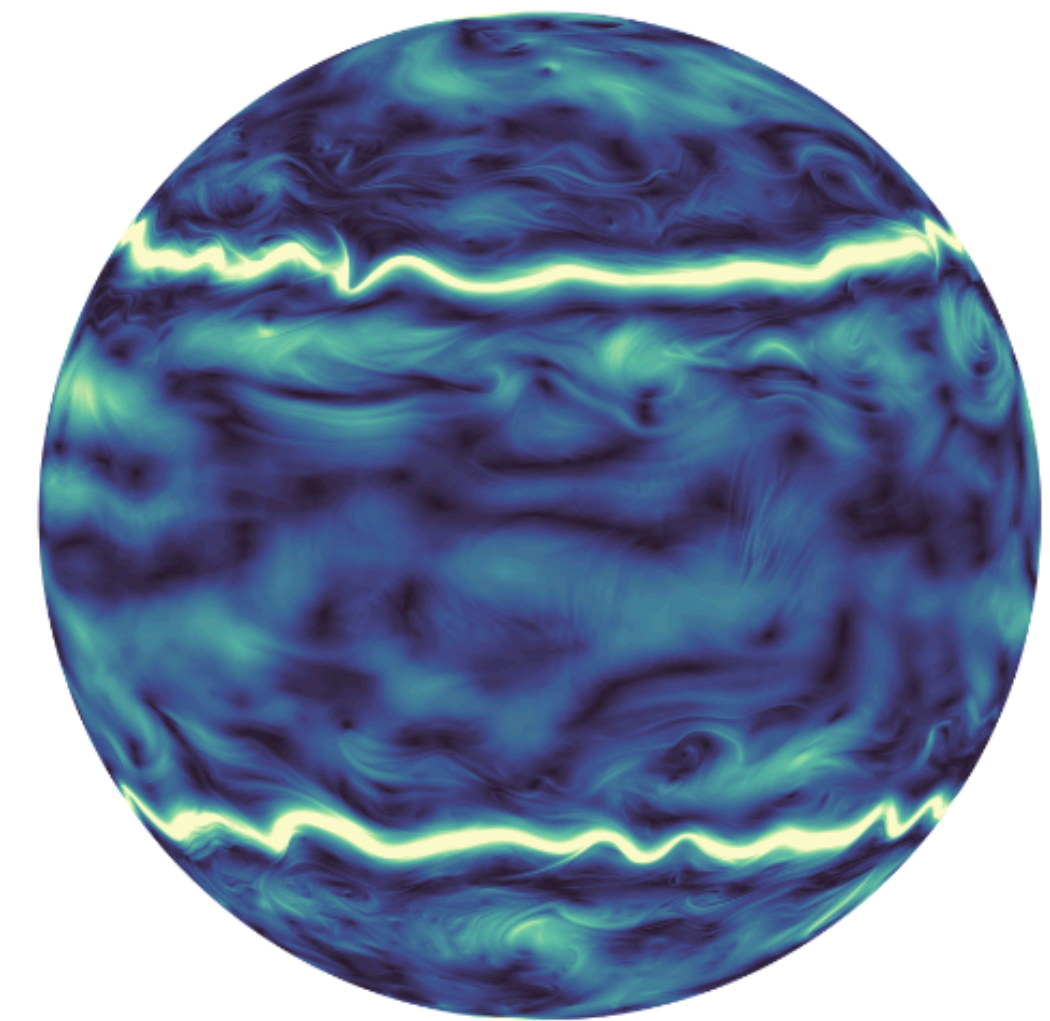
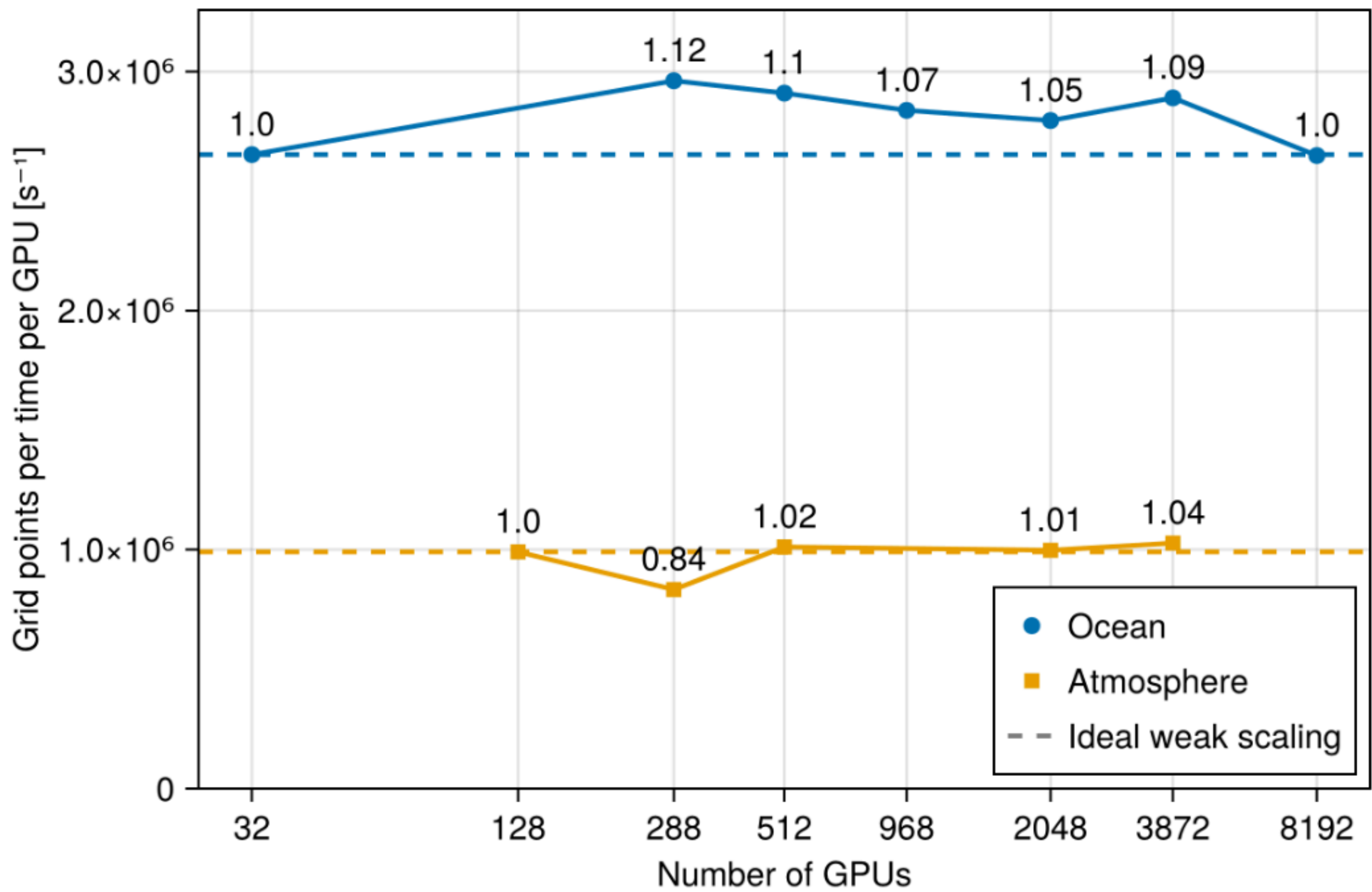
- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
 - Alps (2688 nodes x 4 NVIDIA GH200 GPUs)
 - Perlmutter (1536 nodes x 4 NVIDIA A100 GPUs)
 - 8,192 Google TPUs v7 (4614 F8 TFLOPS each)
- Good Single-Node Perf (CPU)
 - Vanilla Model: 272.0seconds
 - Tensor Optims: 11.5seconds



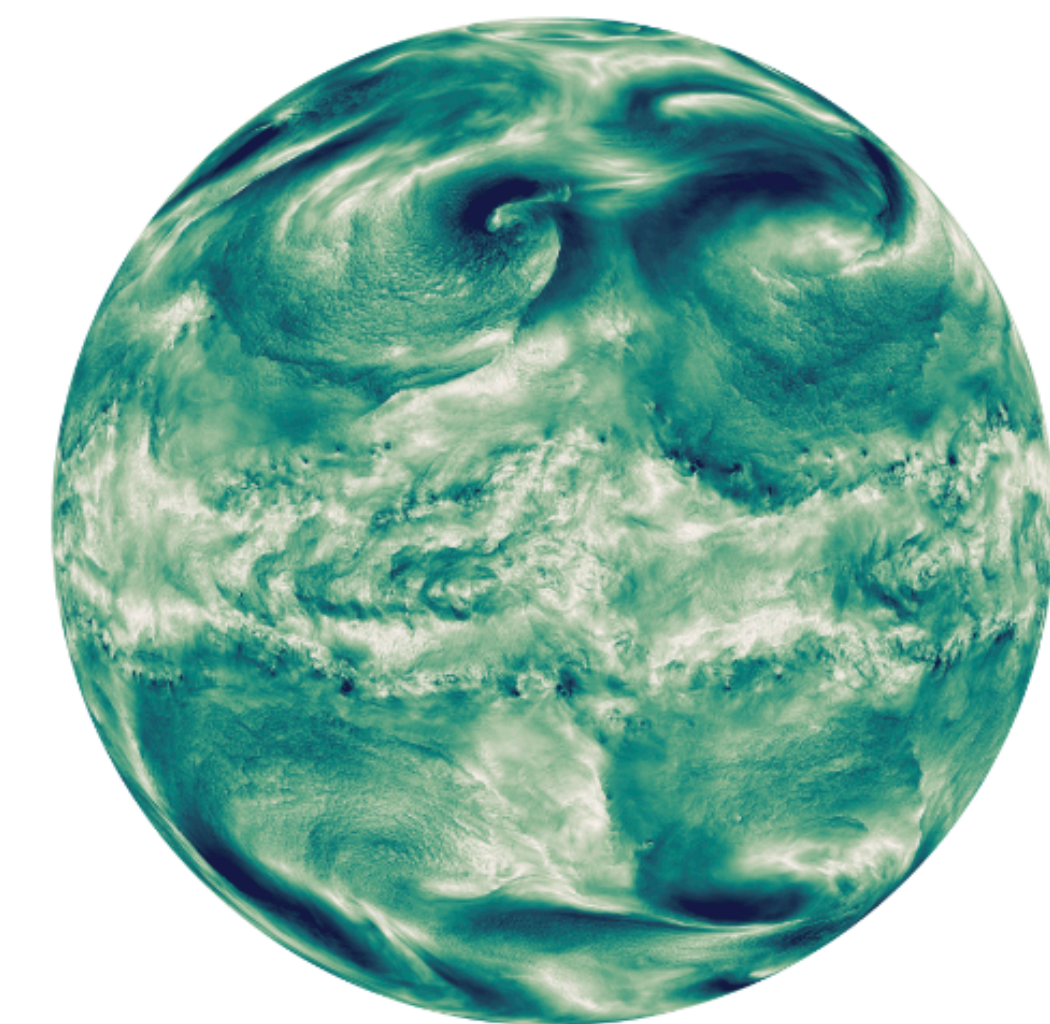
Google Cloud



Performance Results (Alps / NVIDIA GPU)



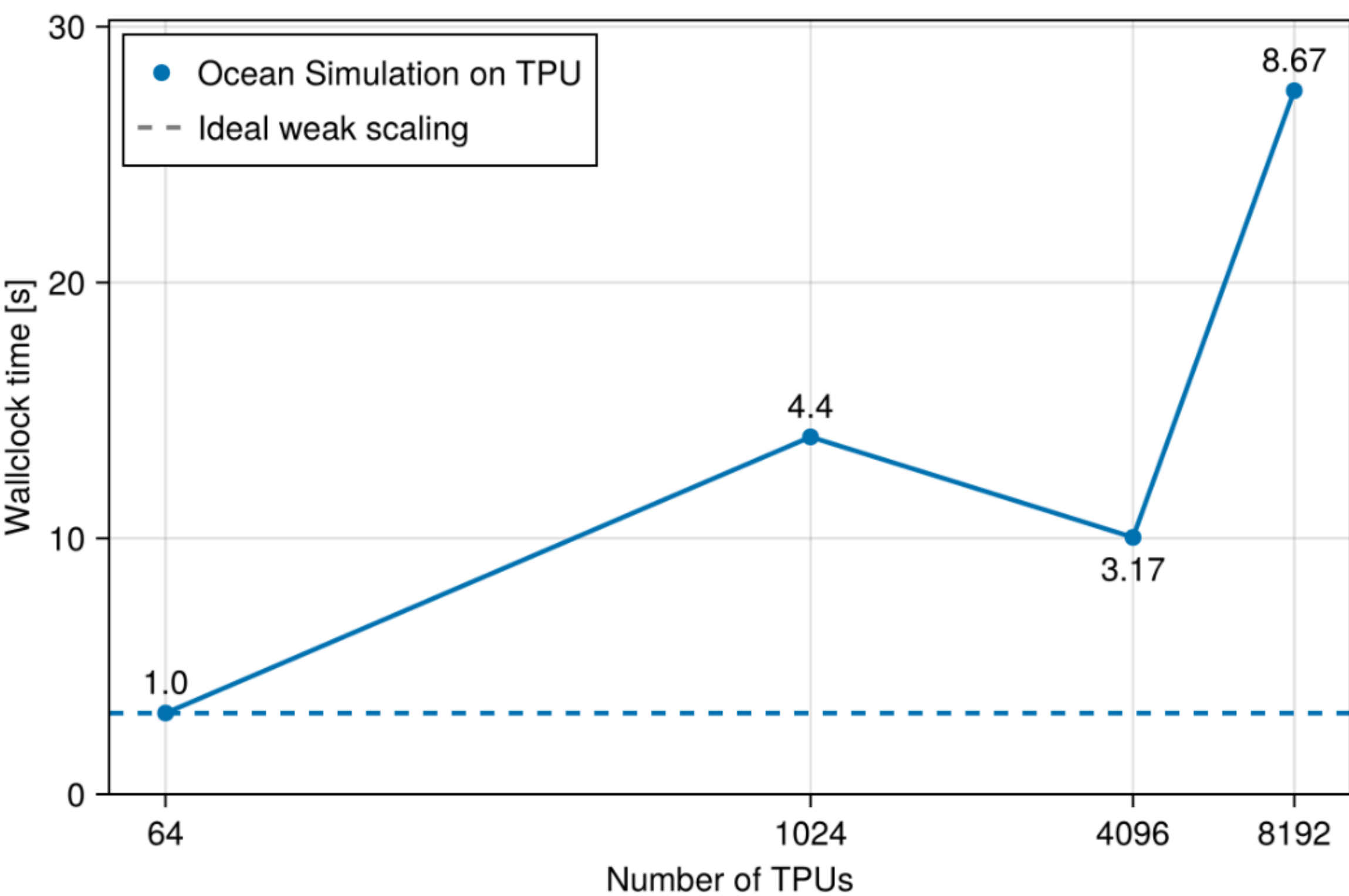
0 1 2 3 4 5 6
Ocean surface speed [m/s]



0 5 10 15 20 25
Atmosphere surface wind speed [m/s]

Performance Results (Google Cloud / TPUs)

- At peak utilized of 97% of the TPU cluster, containing 36 FP8 exaFLOP!
- 1.493 PiB of HBM to store fields!
- 8x128 registers created some additional comms existing patterns don't yet handle (in progress)



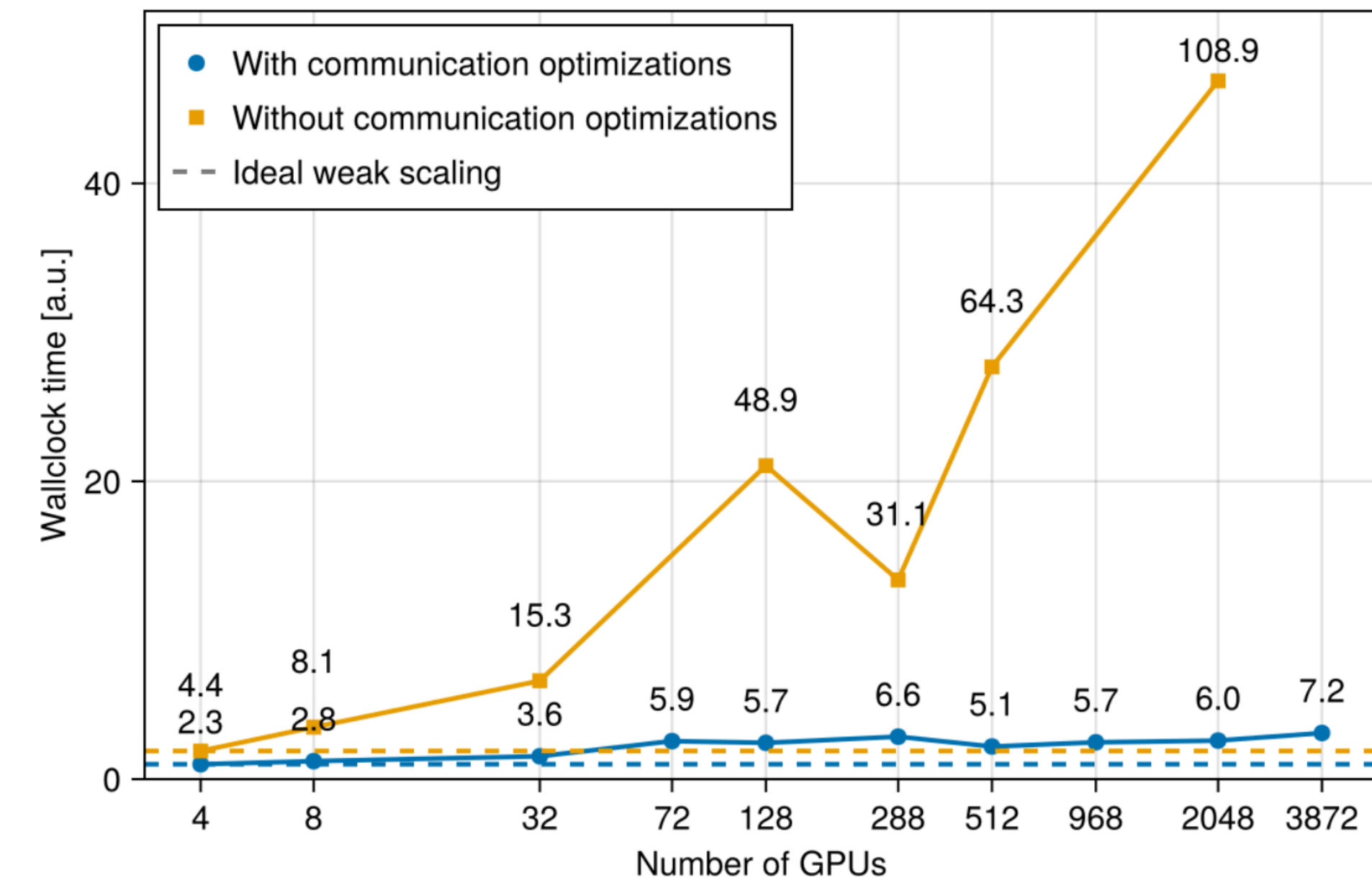
Operation	Percent of Execution
Concatenate	39.04%
Reduce-Window	35.01%
Loop-Fusion 1	19.71%
Data Formatting	2.89%
Slice	1.59%
X64Combine	0.88%
Collective-Permute	0.48%

Distributed + Numerical Optimizations

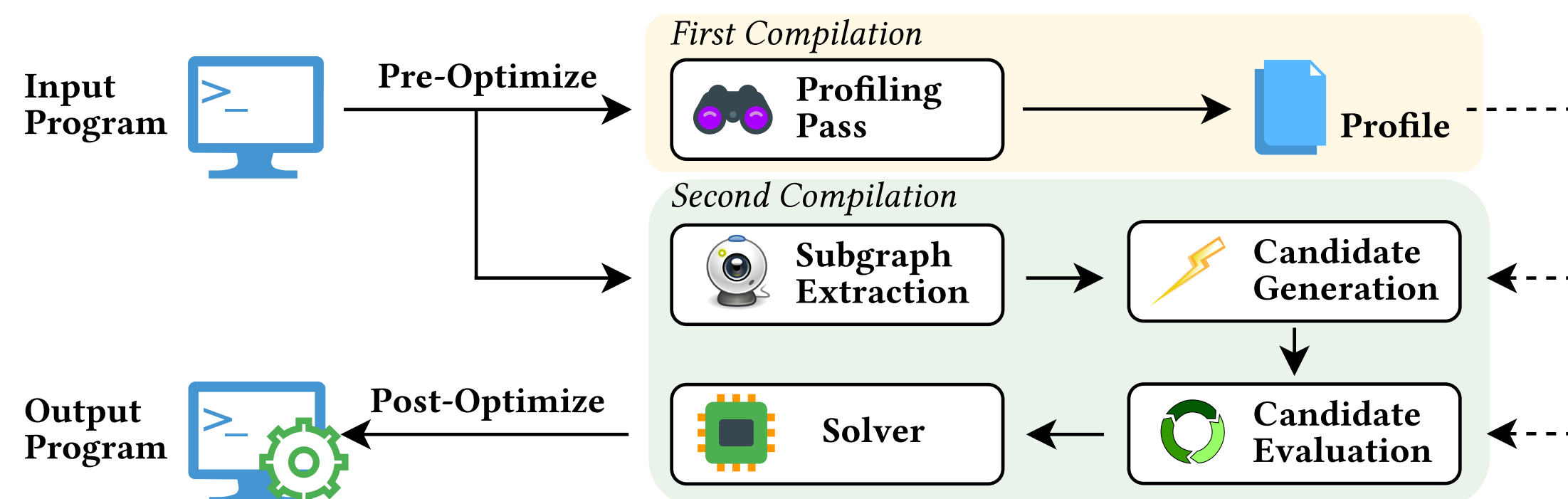
- Novel distributed optimizations improve perf by orders of magnitude (distributed fusion, comm-CSE)

```
# Perform all rotates in a single communication [extending the halo]
%left1, %left2, %left3 = enzymexla.multi_rotate_left %x, 3

%result = %left1 + %left2 + %left3
```



- Can we automatically determine how to map high-level computation to low-precision arithmetic without losing either speed or accuracy?



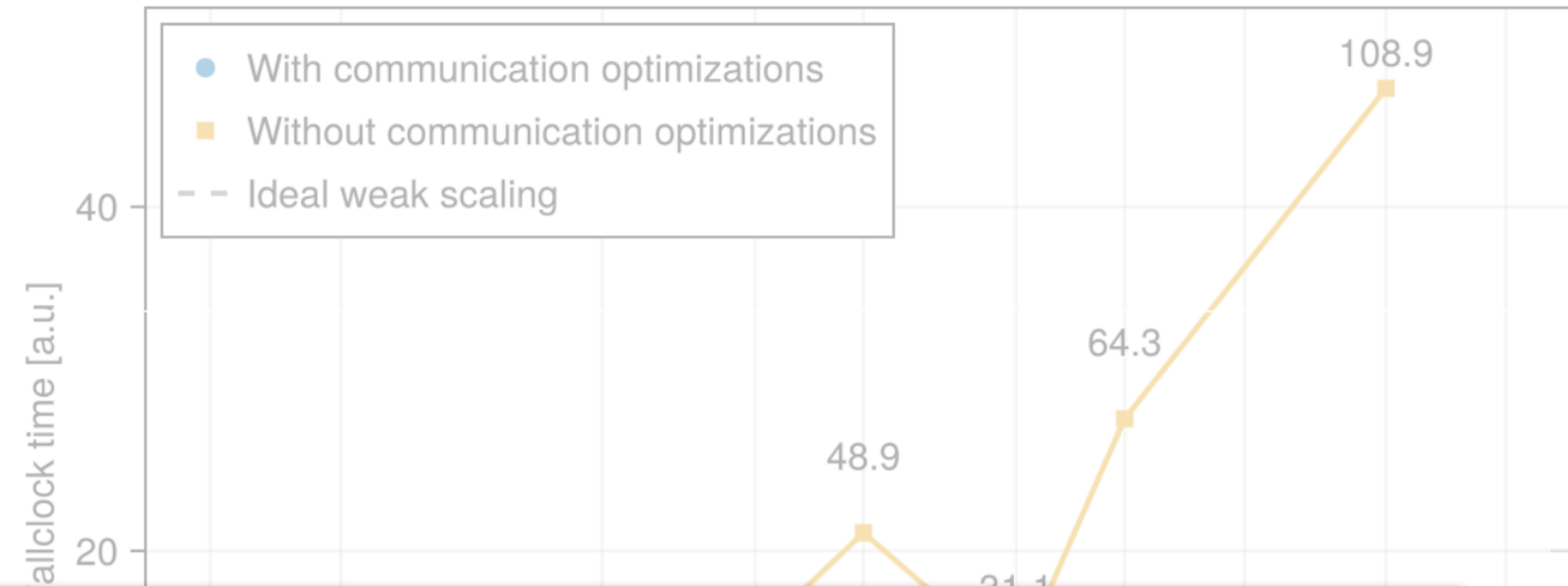
$$\Psi = \frac{\kappa}{2} A^4 - \frac{\mu}{2} \left(J_m \ln \left(1 - \frac{\text{tr}(C) - 3}{J_m} \right) + 2 \ln(J) \right)$$

$$\text{Profiled Bounds: } \left(1 - \frac{\text{tr}(C) - 3}{J_m} \right) \in [0.972, 1.019] (\sim \text{near } 1)$$

$$\text{Discovered Rewrite: } \ln \left(1 - \frac{\text{tr}(C) - 3}{J_m} \right) \rightarrow \log_{1p} \left(-\frac{\text{tr}(C) - 3}{J_m} \right)$$

Distributed + Numerical Optimizations

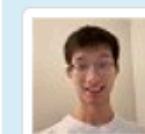
- Novel distributed optimizations improve perf by orders of magnitude (distributed fusion, comm-CSE)



Mon 15 Jun 2026 17:05 - 17:30 at Meadows CD - Session 4

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

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 a plethora of numerical techniques, including algebraic rewriting and mixed-precision tuning, can help navigate the tradeoff between performance and accuracy, applying complex numerical optimizations to real-world code is still a challenging engineering task that requires mathematical expertise and application-specific numerical context. We present Poseidon, a modular and extensible framework that fully automates floating-point optimizations for real-world applications within a production compiler. Within a production compiler, Poseidon auto-extracts symbolic expressions from pre-optimized IR, derives their numerical context from a small surrogate profile, and composes candidates from diverse techniques (e.g., Herbie’s e-graph-based algebraic rewrites and ADAPT-style precision tuning) via a shared dynamic-programming solver. On a quaternion differentiator, Poseidon composes algebraic rewriting with precision tuning for a \$1.46\times\$ speedup at relative error of 10^{-7} . On DOE’s proxy application LULESH, Poseidon recovers optimal FP64 accuracy without substantially reducing performance.



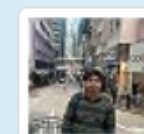
Siyuan Brant · Qian
University of Illinois at Urbana-Champaign



Ivan Ivanov
Institute of Science Tokyo
Japan



Paul Hovland
Argonne National Laboratory
United States



Vimarsh Sathia
University of Illinois Urbana Champaign



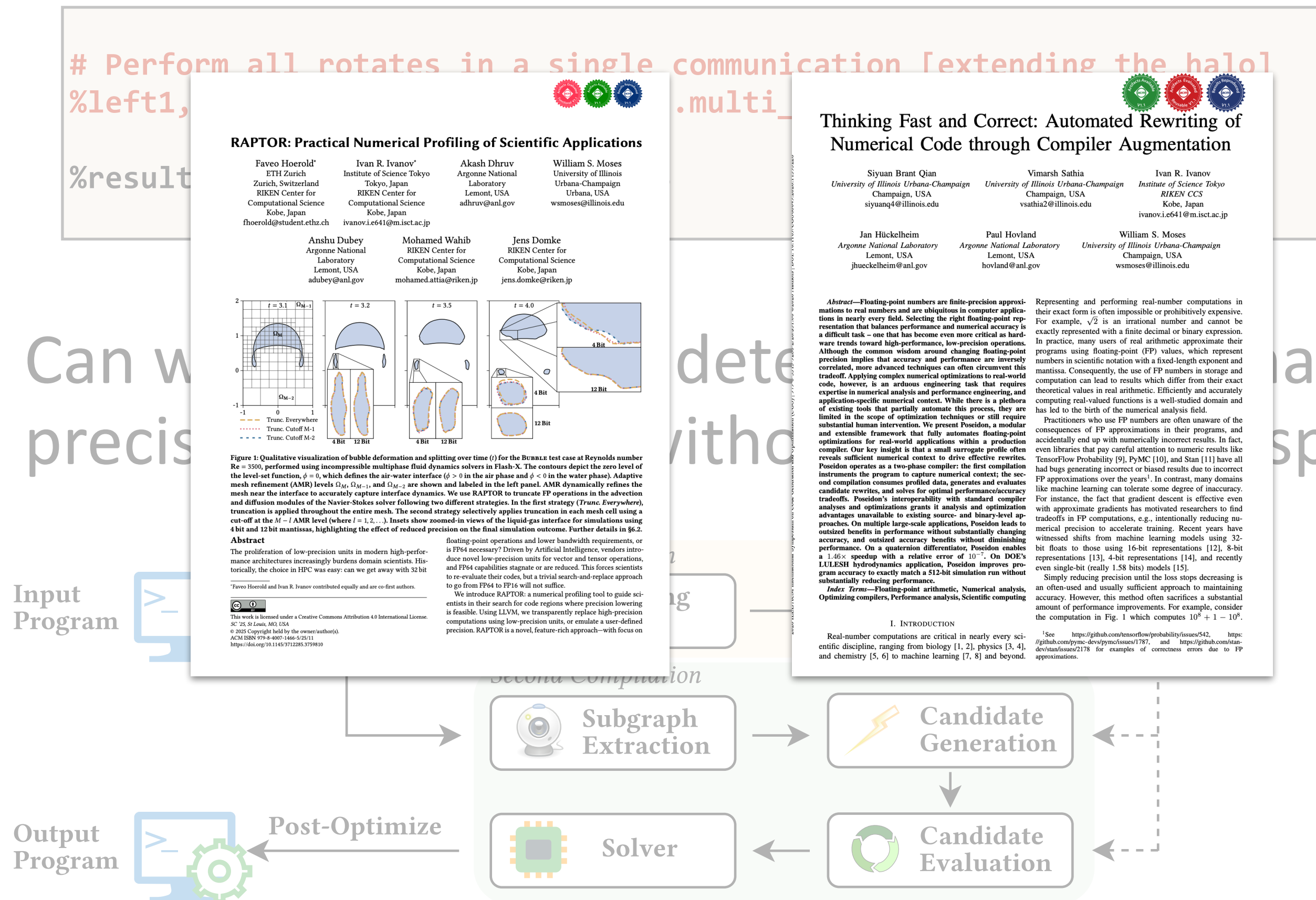
Jan Hueckelheim
Argonne National Laboratory



William S. Moses
University of Illinois Urbana-Champaign
United States

Profiled Bounds: $\left(1 - \frac{1}{J_m}\right) \in [0.972, 1.019]$ ($\sim$ near 1)

Discovered Rewrite: $\ln\left(1 - \frac{\text{tr}(C) - 3}{J_m}\right) \rightarrow \log_{1p}\left(-\frac{\text{tr}(C) - 3}{J_m}\right)$



Conclusions

- Earth system models have four key challenges: scale, data, model, and code
- Compilers are a promising route to automatically resolve all four challenges without requiring the domain scientist become an expert computer scientist
- Demonstrate promising automated compiler rewrites for:
 - Differentiation (Enzyme)
 - Performance Portability (Reactant)
- Works generically on LLVM/MLIR, all open source ([GitHub.com/EnzymeAD](https://github.com/EnzymeAD))
- Lots more problems to solve together, if interested, please reach out!

