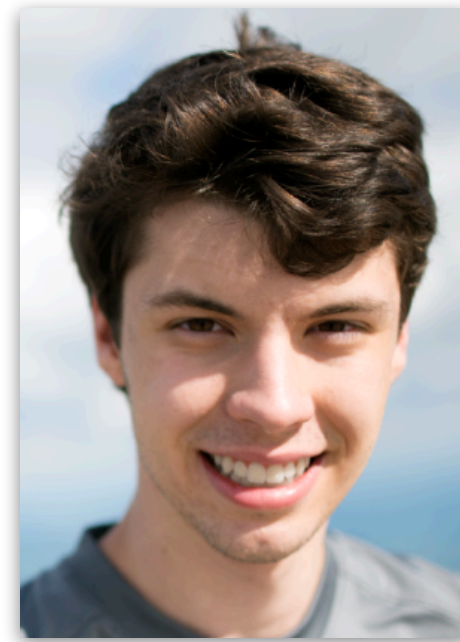
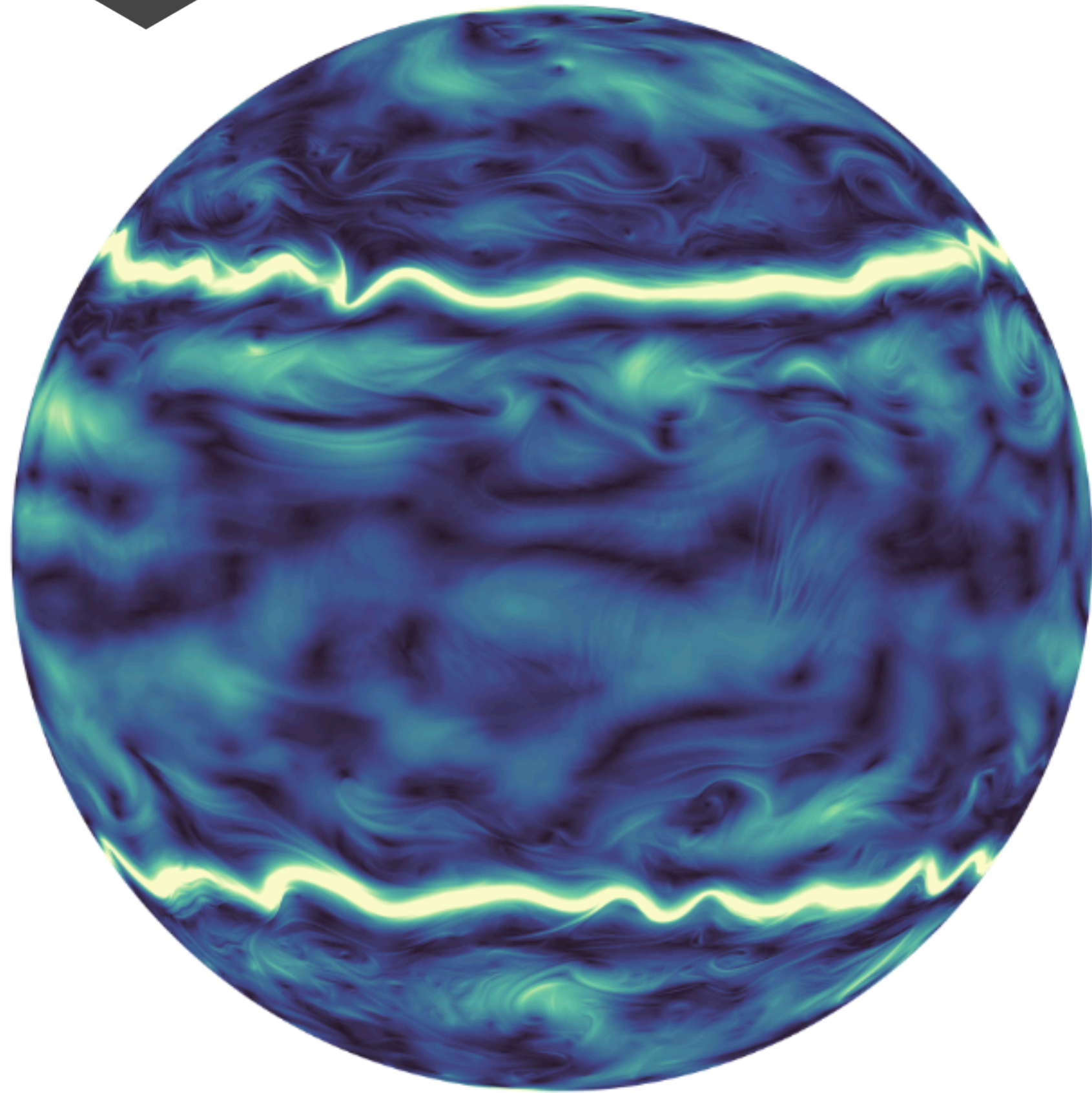
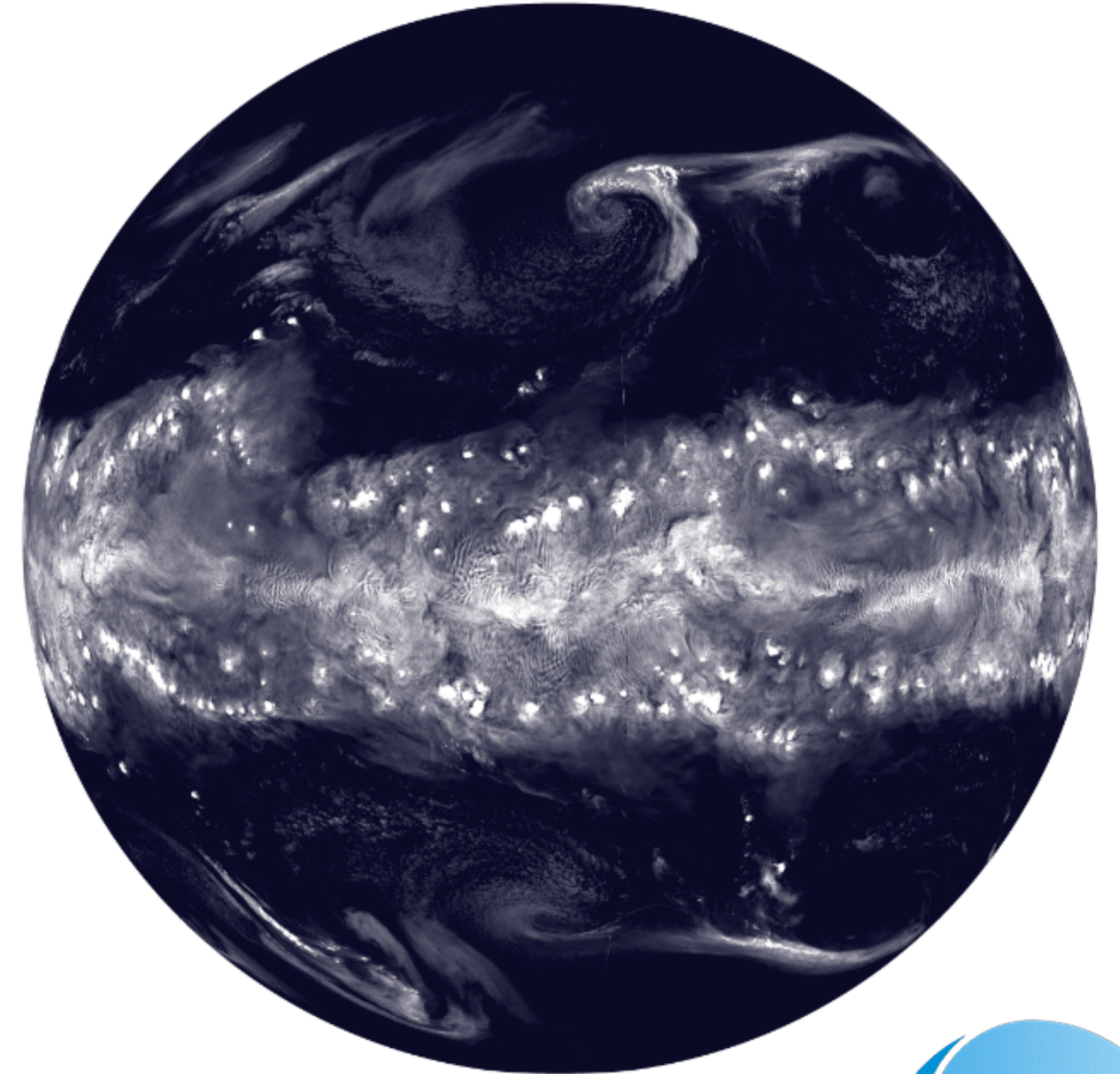




Reactant.jl: Optimize Julia Functions for High Performance on CPU, GPU, TPU



William S. Moses



wsmoses@illinois.edu

PASC '26

July 1, 2026



UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN



RESEARCH ARTICLE
10.1029/2025MS005615

- 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.

Reactant Collaboration
In Submission

DJ4Earth Collaboration
Journal of Atmospheric Modeling
May 2026

The Code I Want To Write != The Code I Want To Run

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```


The Code I Want To Write != The Code I Want To Run

- Allocation for result of $x*x$
 - Matmul computation
- Allocation for result of second $x*x$
 - Matmul computation
- No allocation of transpose (yay types!)
- Allocation for result of add
 - Add computation

```
function f(x: Symmetric)
    return x*x + transpose(x*x)
end
```


The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add

```
function f(x: Symmetric)
    return x*x + transpose(x*x)
end
```


The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add
-
- 2 x Allocs; 1 x Matmuls; 1 x Mul

```
function f(x: Symmetric)
    return x*x + transpose(x*x)
end
```

```
function f2(x: Symmetric)
    return 2*x*x
end
```

The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add

-

```
function f(x: Symmetric)
    return x*x + transpose(x*x)
end
```

- 2 x Allocs; 1 x Matmuls; 1 x Mul

-

```
function f2(x: Symmetric)
    return 2*x*x
end
```

- 1 x Allocs; 1 x Matmuls

```
function f3(x: Symmetric)
    out = similar(x)
    mul!(out, x, x, 2, 0)
end
```


Why Can't the Compiler Fix this for Me?

```
function f(x)
  return x*x + transpose(x*x)
end
```

```
@code_typed f(x)
```

```
CodeInfo(
```

```
1 — %1 = invoke Main.:*(x::Matrix{Float64}, x::Matrix{Float64})::Matrix{Float64}
```

```
|   %2 = invoke Main.:*(x::Matrix{Float64}, x::Matrix{Float64})::Matrix{Float64}
```

```
|   %3 = %new(Transpose{Float64, Matrix{Float64}}, %2)::Transpose{Float64, Matrix{Float64}}
```

```
|   %4 = invoke Main.:+(%1::Matrix{Float64}, %3::Transpose{Float64, Matrix{Float64}})::Matrix{Float64}
```

```
|   └─ return %4
```

```
) => Matrix{Float64}
```

@code_typed y*y

```
CodeInfo(
1  ——— %1  = Base.arraysize(A, 1)::Int64
  |      %2  = Base.arraysize(B, 2)::Int64
  |      %3  = $(Expr(:foreigncall, :(:jl_alloc_array_2d), Matrix{Float64}, svec(Any, Int64, Int64), 0, :(:ccall), Matrix{Float64}, :(%1), :(%2), :(%2), :(%1)))::Matrix{Float64}
  |_____ goto #25 if not true
2  ----- %5  = ϕ (#1 => 'N', #23 => %47)::Char
  |      %6  = ϕ (#1 => 2, #23 => %48)::Int64
  |_____ goto #13 if not true
3  ----- %8  = ϕ (#2 => 'N', #12 => %26)::Char
  |      %9  = ϕ (#2 => 2, #12 => %27)::Int64
  |      %10 = Base.bitcast(Base.UInt32, %8)::UInt32
  |      %11 = Base.bitcast(Base.UInt32, %5)::UInt32
  |      %12 = (%10 === %11)::Bool
  |_____ goto #5 if not %12
4  ———      goto #14
5  ——— %15 = Base.sle_int(1, %9)::Bool
  |_____ goto #7 if not %15
6  ——— %17 = Base.sle_int(%9, 3)::Bool
  |_____ goto #8
7  ———      nothing::Nothing
8  ----- %20 = ϕ (#6 => %17, #7 => false)::Bool

...

24 —      goto #26
25 ---      goto #26
26 --- %55 = ϕ (#24 => false, #25 => true)::Bool
  |_____ goto #27
27 —      goto #33 if not %55
28 —      goto #30 if not true
29 —      nothing::Nothing
30 ---      goto #32 if not true
31 —      nothing::Nothing
32 --- %62 = invoke LinearAlgebra.gemm_wrapper!(%3::Matrix{Float64}, 'N'::Char, 'N'::Char, A::Matrix{Float64}, B::Matrix{Float64}, $(QuoteNode(LinearAlgebra.MulAddMul{true, true, Bool, Bool})(true, false)))::LinearAlgebra.MulAddMul{true, true, Bool, Bool}::Matrix{Float64}
  |_____ goto #37
33 —      goto #36 if not true
34 —      goto #36 if not true
35 —      nothing::Nothing
36 --- %67 = invoke LinearAlgebra._generic_matmatmul!(%3::Matrix{Float64}, 'N'::Char, 'N'::Char, A::Matrix{Float64}, B::Matrix{Float64}, $(QuoteNode(LinearAlgebra.MulAddMul{true, true, Bool, Bool})(true, false)))::LinearAlgebra.MulAddMul{true, true, Bool, Bool}::Matrix{Float64}
  |_____ goto #37
37 --- %69 = ϕ (#32 => %62, #36 => %67)::Matrix{Float64}
  |_____ goto #38
38 —      goto #39
39 —      return %69
) => Matrix{Float64}
```


What is MLIR?



Multi-Level Intermediate Representation (MLIR)

- New Compiler IR with user-defined instructions, optimizations, analyses
 - Linear Algebra
 - GPU Programming
 - Fully Homomorphic Encryption
- Mix and match dialects and optimizations from multiple dialects
- Core infrastructure of modern ML frameworks (JaX, PyTorch, TensorFlow)
- Frontends for C++ (Polygeist), Julia (Reactant.jl, this talk :)

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
  affine.for %arg2 = 0 to 10 {  
    affine.store %arg1, %arg0 [2 * %arg2] : memref<?xi32>  
  }  
  return  
}
```

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

$\text{transpose}(\text{transpose}(x)) \rightarrow x$

$\text{transpose}(\text{matmul}(a, b)) \rightarrow$
 $\text{matmul}(b, a)$

Often require program context

$\text{transpose}(\text{convert}(\text{reshape}(x)))$
 $\leftrightarrow \text{reshape}(\text{convert}(\text{transpose}(x)))$

$\text{slice}(\text{add}(a, b)) \rightarrow$
 $\text{add}(\text{slice}(a), \text{slice}(b))$

$\text{mul}(\text{pad}(x, 0), y) \rightarrow$
 $\text{pad}(\text{mul}(x, \text{slice}(y)), 0)$

```
x, y : tensor<100000xf32>
```

```
a = dot(x, y)
```

```
b = mul(a, z)
```

```
c = add(b, 4)
```

```
return c[0:10]
```


Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

$\text{transpose}(\text{transpose}(x)) \rightarrow x$

$\text{transpose}(\text{matmul}(a, b)) \rightarrow$
 $\text{matmul}(b, a)$

Often require program context

$\text{transpose}(\text{convert}(\text{reshape}(x)))$
 $\leftrightarrow \text{reshape}(\text{convert}(\text{transpose}(x)))$

$\text{slice}(\text{add}(a, b)) \rightarrow$
 $\text{add}(\text{slice}(a), \text{slice}(b))$

$\text{mul}(\text{pad}(x, 0), y) \rightarrow$
 $\text{pad}(\text{mul}(x, \text{slice}(y)), 0)$

```
x, y : tensor<100000xf32>
```

```
a = dot(x, y)
```

```
b = mul(a, z)
```

```
c = add(b[0:10], 4)
```

```
return c
```

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

$\text{transpose}(\text{transpose}(x)) \rightarrow x$

$\text{transpose}(\text{matmul}(a, b)) \rightarrow$
 $\text{matmul}(b, a)$

Often require program context

$\text{transpose}(\text{convert}(\text{reshape}(x)))$
 $\leftrightarrow \text{reshape}(\text{convert}(\text{transpose}(x)))$

$\text{slice}(\text{add}(a, b)) \rightarrow$
 $\text{add}(\text{slice}(a), \text{slice}(b))$

$\text{mul}(\text{pad}(x, 0), y) \rightarrow$
 $\text{pad}(\text{mul}(x, \text{slice}(y)), 0)$

```
x, y : tensor<100000xf32>
```

```
a = dot(x, y)
```

```
b = mul(a[0:10], z[0:10])
```

```
c = add(b, 4)
```

```
return c
```

Performance Engineering a Toy LLM

- Introduction of linear-algebra specific optimizations made a significant impact on training performance
 - 53% speedup of a run with 32x accelerators
 - 14-18.5% speedup for single accelerator
- Accessible now from github.com/EnzymeAD/Enzyme-JaX

```
Step: 2 loss: 12.880576133728027
Step: 3 loss: 12.785988807678223
Step: 4 loss: 12.652521133422852
Step: 5 loss: 12.482083320617676
...
Step: 19 loss: 9.190348625183105
Step: 20 loss: 8.928218841552734
Step: 21 loss: 8.660679817199707
```

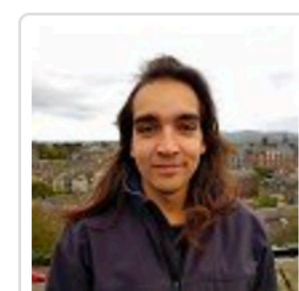
Unopt: 0.479 samples/sec

```
Step: 2 loss: 12.88175106048584
Step: 3 loss: 12.786417007446289
Step: 4 loss: 12.652612686157227
Step: 5 loss: 12.482114791870117
...
Step: 19 loss: 9.189879417419434
Step: 20 loss: 8.929146766662598
Step: 21 loss: 8.659639358520508
```

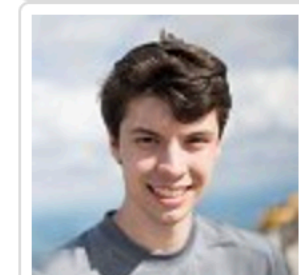
Opt: 0.736 samples/sec

Performance Engineering a Toy LLM

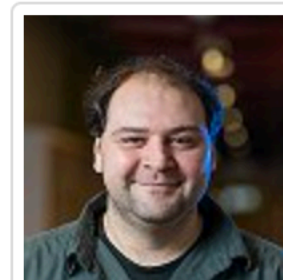
- Introduction of linear-algebra specific optimizations made a significant impact on training performance
 - 53% speedup of a run with 32x accelerators
 - 14-18.5% speedup for single accelerator
- Accessible now from github.com/EnzymeAD/Enzyme-JaX



Martin Lücke
University of Edinburgh
United Kingdom



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



Michel Steuwer
Technische Universität
Berlin
Germany



Alex Zinenko
Google DeepMind



Albert Cohen
Google DeepMind
France

Mon 3 Mar

Displayed time zone: **Pacific Time (US & Canada)** [change](#)

17:40

20m



The MLIR Transform Dialect - Your compiler is more powerful than you think

Talk

Martin Lücke University of Edinburgh, Michel Steuwer Technische Universität Berlin, Albert Cohen Google DeepMind, William S. Moses University of Illinois Urbana-Champaign, Alex Zinenko Google DeepMind

```
Step: 2 loss: 12.880576133728027
Step: 3 loss: 12.785988807678223
Step: 4 loss: 12.652521133422852
Step: 5 loss: 12.482083320617676
...
Step: 19 loss: 9.190348625183105
Step: 20 loss: 8.928218841552734
Step: 21 loss: 8.660679817199707
```

Unopt: 0.479 samples/sec

```
Step: 2 loss: 12.88175106048584
Step: 3 loss: 12.786417007446289
Step: 4 loss: 12.652612686157227
Step: 5 loss: 12.482114791870117
...
Step: 19 loss: 9.189879417419434
Step: 20 loss: 8.929146766662598
Step: 21 loss: 8.659639358520508
```

Opt: 0.736 samples/sec

Reactant.jl

- Optimizing Compiler Framework For Julia, built on top of MLIR
- Preserves high-level structures from Julia code into the compiler
 - Effectively optimize programs
 - Effectively retarget programs (allowing them to run on distributed clusters of your favorite accelerator, including CPU/GPU/TPU)
- Can leverage XLA (state of the art runtime, used by TensorFlow, JaX, & PyTorch)
- Compatible with mutation, control flow(*), autodiff (Enzyme)
- 17 • All open source ([GitHub.com/EnzymeAD/Reactant.jl](https://github.com/EnzymeAD/Reactant.jl))



Reactant.jl Usage

```
using Reactant

x = Reactant.to_rarray(ones(3,3))

function f(x)
    return x*x + transpose(x*x)
end

r_f = @compile f(x)

r_f(x)
```

- Reactant has 3 core primitives:
 - Reactant.ConcreteArray
 - @compile
 - Reactant.TracedRArray

Reactant.jl ConcreteArrays

```
julia> x = Reactant.to_rarray(ones(3,3))
```

```
2025-07-25 09:09:49.875290: I external/xla/xla/service/service.cc:163] XLA  
service 0x11e08f0 initialized for platform CUDA (this does not guarantee that  
XLA will be used).
```

```
julia> r_f(x)
```

```
3×3 ConcretePJRTArray{Float64,2}:
```

```
 6.0  6.0  6.0
```

```
 6.0  6.0  6.0
```

```
 6.0  6.0  6.0
```

```
julia> Reactant.devices()
```

```
2-element Vector{Reactant.XLA.PJRT.Device}:
```

```
 Reactant.XLA.PJRT.Device{Ptr{Nothing} @0x00000000013d21a0, "CUDA:0 NVIDIA  
GeForce RTX 5090")
```

```
 Reactant.XLA.PJRT.Device{Ptr{Nothing} @0x0000000000aefd90, "CUDA:1 NVIDIA  
GeForce RTX 5090")
```



Reactant.jl Compilation

```
julia> @compile optimize=false f(x)
```

```
module @reactant_f attributes {mhlo.num_partitions = 1 : i64, mhlo.num_replicas = 1 : i64} {
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_1"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_2"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_3"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"+_broadcast_scalar"(%arg0: tensor<f64>, %arg1: tensor<f64>) -> (tensor<f64>, tensor<f64>, tensor<f64>) {
    %0 = stablehlo.add %arg0, %arg1 : tensor<f64>
    return %0, %arg0, %arg1 : tensor<f64>, tensor<f64>, tensor<f64>
  }
  func.func @main(%arg0: tensor<3x3xf64> {tf.aliasing_output = 1 : i32}) -> (tensor<3x3xf64>, tensor<3x3xf64>) {
    %0 = stablehlo.transpose %arg0, dims = [1, 0] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %1 = stablehlo.convert %0 : tensor<3x3xf64>
    %2 = stablehlo.convert %0 : tensor<3x3xf64>
    %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %3 = stablehlo.broadcast_in_dim %1, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %4 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar"(%3) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %5 = stablehlo.convert %4 : tensor<3x3xf64>
    %cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %6 = stablehlo.broadcast_in_dim %2, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %7 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_1"(%6) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %8 = stablehlo.convert %7 : tensor<3x3xf64>
    %9 = stablehlo.dot_general %5, %8, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>, tensor<3x3xf64>) -> tensor<3x3xf64>
    %cst_2 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %10 = stablehlo.convert %0 : tensor<3x3xf64>
    %11 = stablehlo.convert %0 : tensor<3x3xf64>
    %cst_3 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %12 = stablehlo.broadcast_in_dim %10, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %13 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_2"(%12) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %14 = stablehlo.convert %13 : tensor<3x3xf64>
    %cst_4 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %15 = stablehlo.broadcast_in_dim %11, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %16 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_3"(%15) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %17 = stablehlo.convert %16 : tensor<3x3xf64>
    %18 = stablehlo.dot_general %14, %17, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>, tensor<3x3xf64>) -> tensor<3x3xf64>
  }
}
```



Reactant.jl Compilation

```
julia> @compile f(x)
module @reactant_f attributes {mhlo.num_partitions = 1 : i64, mhlo.num_replicas = 1 : i64} {
  func.func @main(%arg0: tensor<3x3xf64>) -> tensor<3x3xf64> {
    %0 = stablehlo.dot_general %arg0, %arg0, contracting_dims = [0] x [1], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>,
tensor<3x3xf64>) -> tensor<3x3xf64>
    %1 = stablehlo.transpose %0, dims = [1, 0] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %2 = stablehlo.add %1, %0 : tensor<3x3xf64>
    return %2 : tensor<3x3xf64>
  }
}

julia> @code_xla f(x)
HloModule reactant_f, is_scheduled=true, entry_computation_layout={(f64[3,3]{1,0})->f64[3,3]{1,0}},
frontend_attributes={fingerprint_before_lhs="e0828ffd4833737b459c41de309ba417"}

% fused_add (param_0.1: f64[3,3]) -> f64[3,3] {
  %param_0.1 = f64[3,3]{1,0} parameter(0)
  %transpose.2 = f64[3,3]{1,0} transpose(%param_0.1), dimensions={1,0}
  ROOT %add.4.1 = f64[3,3]{1,0} add(%transpose.2, %param_0.1), metadata={op_name="add" source_file="/mnt3/wmoses/git/Reactant.jl/
src/Ops.jl" source_line=375}
}

ENTRY %main.5 (Arg_0.1: f64[3,3]) -> f64[3,3] {
  %Arg_0.1 = f64[3,3]{1,0} parameter(0), metadata={op_name="arg1 (path=(args, 1))"}
  %custom-call.1 = (f64[3,3]{1,0}, s8[144]{0}) custom-call(%Arg_0.1, %Arg_0.1), custom_call_target="__cublas$gemm",
metadata={op_name="dot_general" source_file="/mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=808},
backend_config={"operation_queue_id":"0","wait_on_operation_queues":[],"gemm_backend_config":
{"alpha_real":1,"beta":0,"dot_dimension_numbers":{"lhs_contracting_dimensions":["0"],"rhs_contracting_dimensions":
["1"],"lhs_batch_dimensions":[],"rhs_batch_dimensions":[]},"alpha_imag":0,"precision_config":{"operand_precision":
["DEFAULT","DEFAULT"],"algorithm":"ALG_UNSET"},"epilogue":"DEFAULT","lhs_stride":"9","rhs_stride":"9","grad_x":false,"grad_y":false,
"damax_output":false},"force_earliest_schedule":false,"reification_cost":[]}
  %get-tuple-element.1 = f64[3,3]{1,0} get-tuple-element(%custom-call.1), index=0, metadata={op_name="dot_general" source_file="/
mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=808}
  ROOT %loop_add_fusion = f64[3,3]{1,0} fusion(%get-tuple-element.1), kind=kLoop, calls=%fused_add, metadata={op_name="add"
source_file="/mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=375}
}
```



Reactant.jl TracedArrays

- ConcreteArray : Reactant-specific array type (which could have information stored across multiple devices)
- TracedArray: Reactant-specific array type used during compilation which represent data whose values are not compile-time constants

```
julia> function g(x)
    @show x
    return x*x + transpose(x*x)
end

julia> r_g = @compile g(x)
x = TracedRArray{Float64,2N}(((args, 1),), size=(3, 3))
```



Reactant.jl TracedArrays

```
julia> function h(cond, x)
    if cond
        return x
    else
        return x*x
    end
end
```

h (generic function with 2 methods)

```
julia> cond = Reactant.to_rarray(true, track_numbers=Number)
ConcretePJRTNumber{Bool}(true)
```

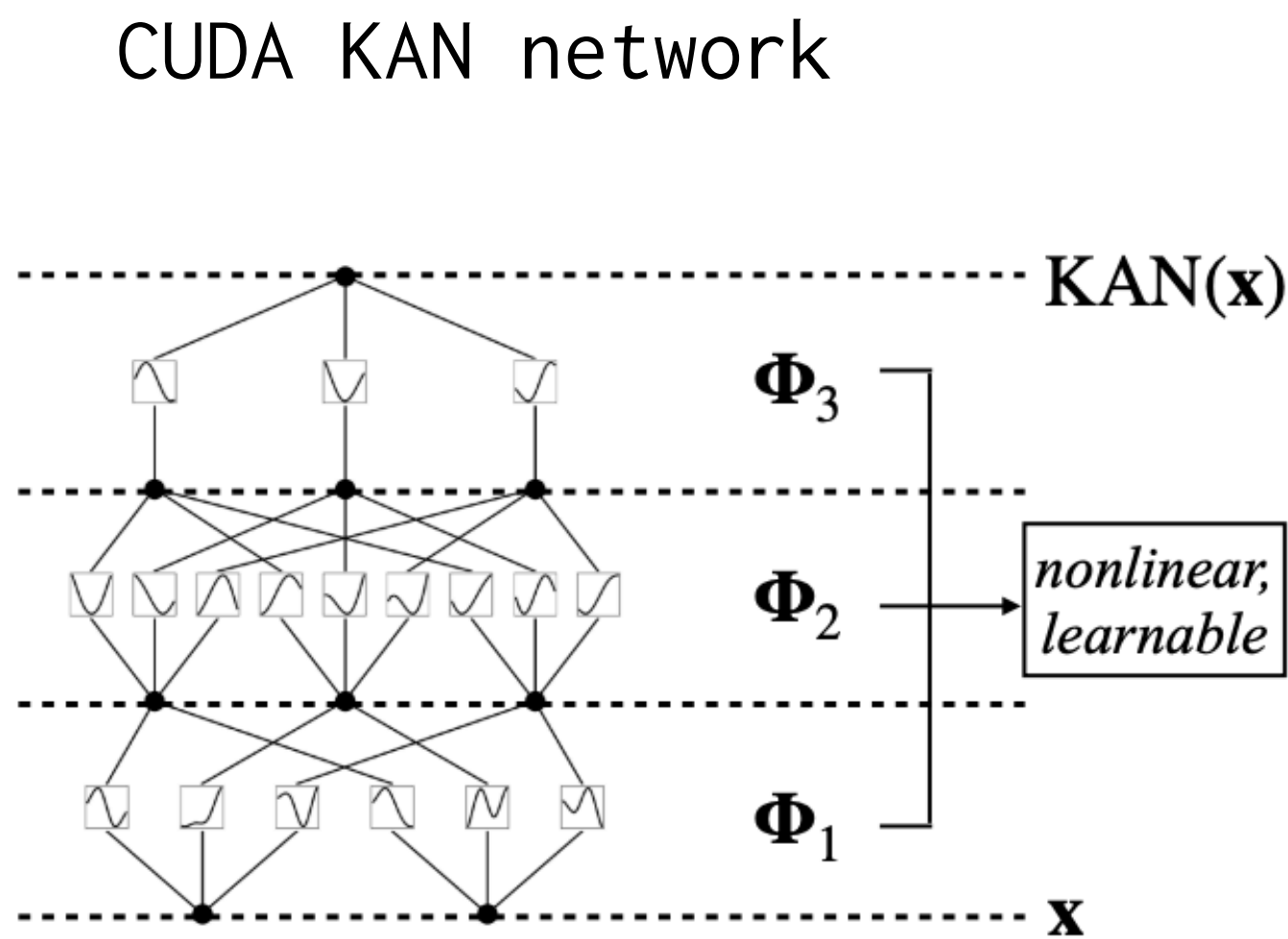
```
julia> @jit h(cond, x)
ERROR: TypeError: non-boolean (Reactant.TracedRNumber{Bool}) used in boolean context
```

```
julia> function h2(cond, x)
    @trace if cond
        return x
    else
        return x*x
    end
end
```

h2 (generic function with 1 method)

```
julia> @jit h2(cond, x)
1-element Vector{ConcretePJRTArray{Float64, 2, 1}}:
 ConcretePJRTArray{Float64, 2, 1}([1.0 1.0 1.0; 1.0 1.0 1.0; 1.0 1.0 1.0])
```


EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)



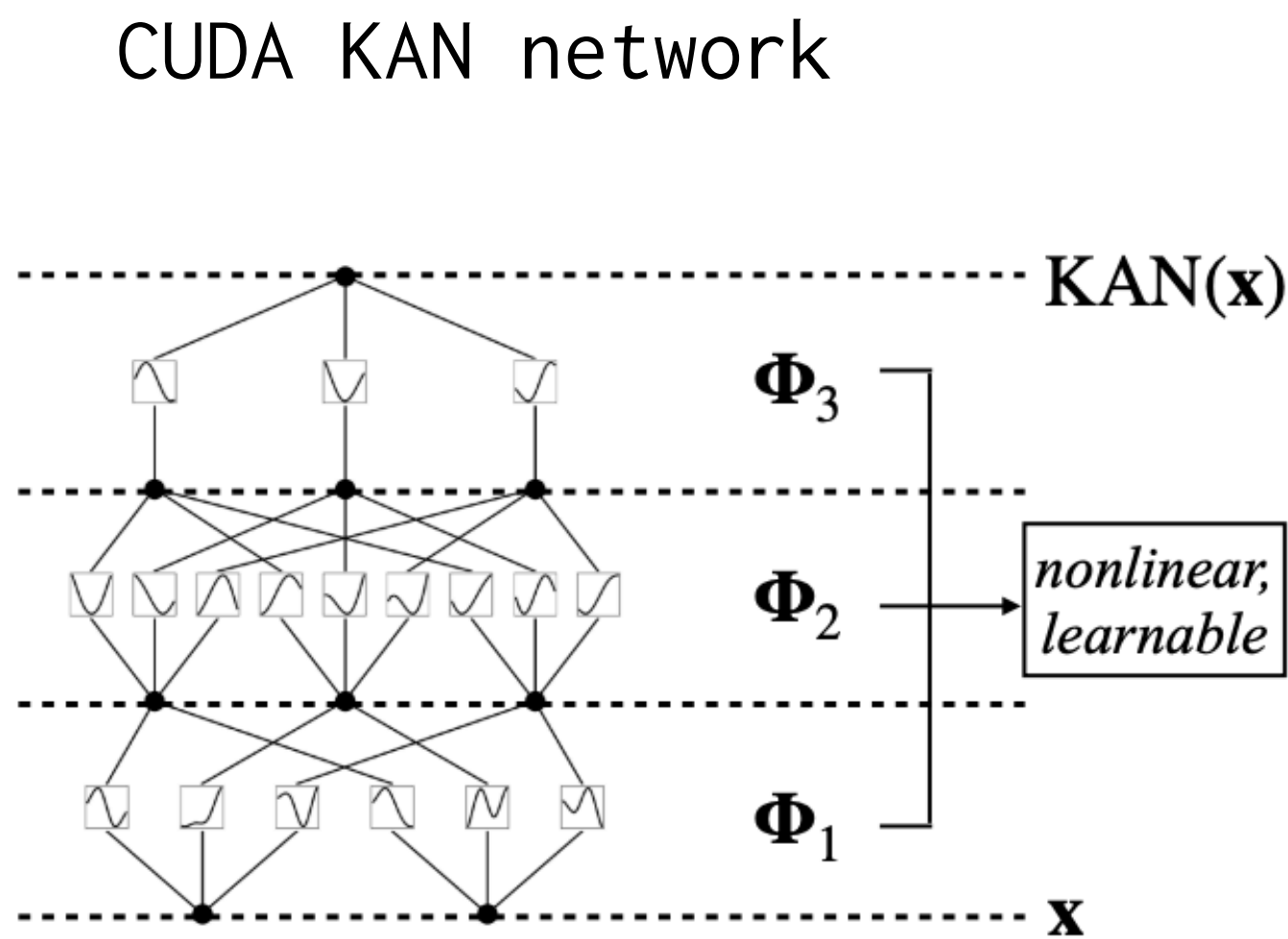
Forward (regular Julia)
47.586 us (248 allocations)
234.233 us (1022 allocations)
134.028 us (668 allocations)

Backwards (Zygote + Julia)
289.319 us (575 allocations)
2099.000 us (1055 allocations)
1772.000 us (877 allocations)

Forward (Reactant)
39.873 us (2 allocations)
68.439 us (6 allocations)
55.889 us (6 allocations)

Backwards (EnzymeMLIR + Reactant)
51.691 us (3 allocations)
104.193 us (3 allocations)
80.020 us (3 allocations)

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)



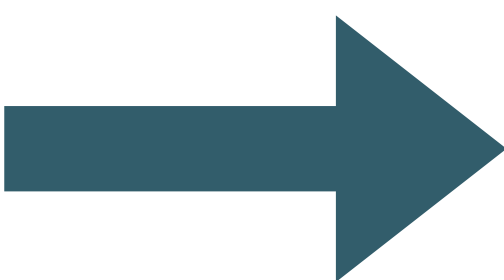
Forward (regular Julia)
47.586 us (248 allocations)
234.233 us (1022 allocations)
134.028 us (668 allocations)

Forward (Reactant)
39.873 us (2 allocations)
68.439 us (6 allocations)
55.889 us (6 allocations)

Backwards (Zygote + Julia)
289.319 us (575 allocations)
2099.000 us (1055 allocations)
1772.000 us (877 allocations)

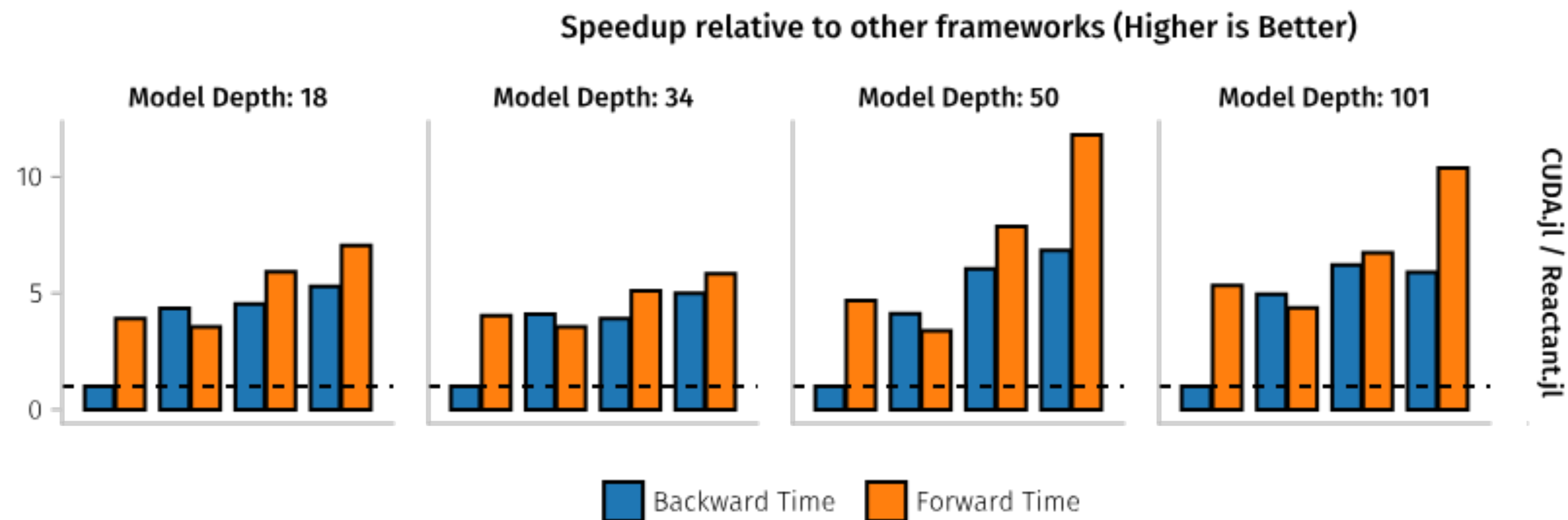
Backwards (EnzymeMLIR + Reactant)
51.691 us (3 allocations)
104.193 us (3 allocations)
80.020 us (3 allocations)

2.14x speedup
(Primal)



13.57x speedup
(Derivative)

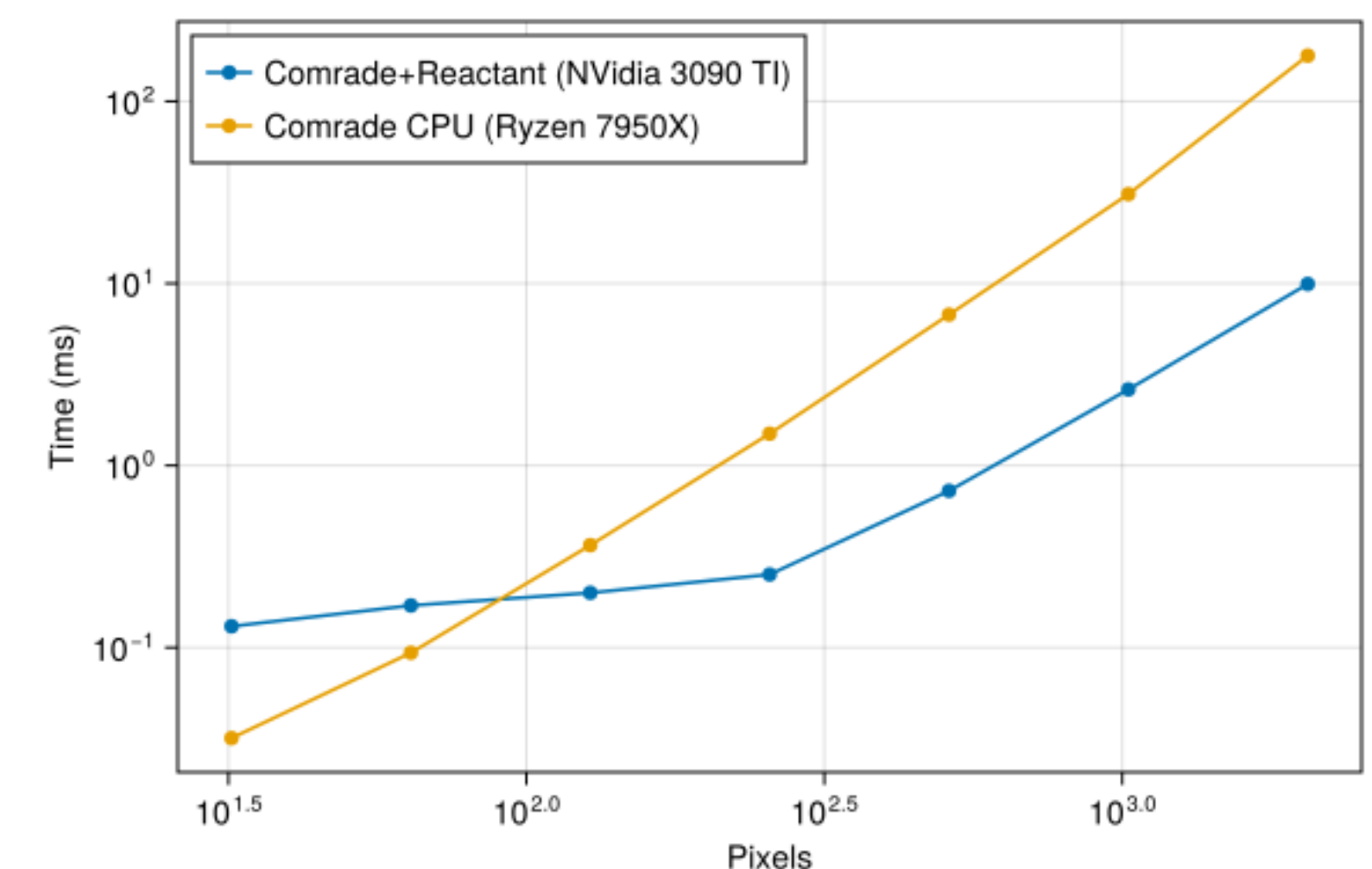
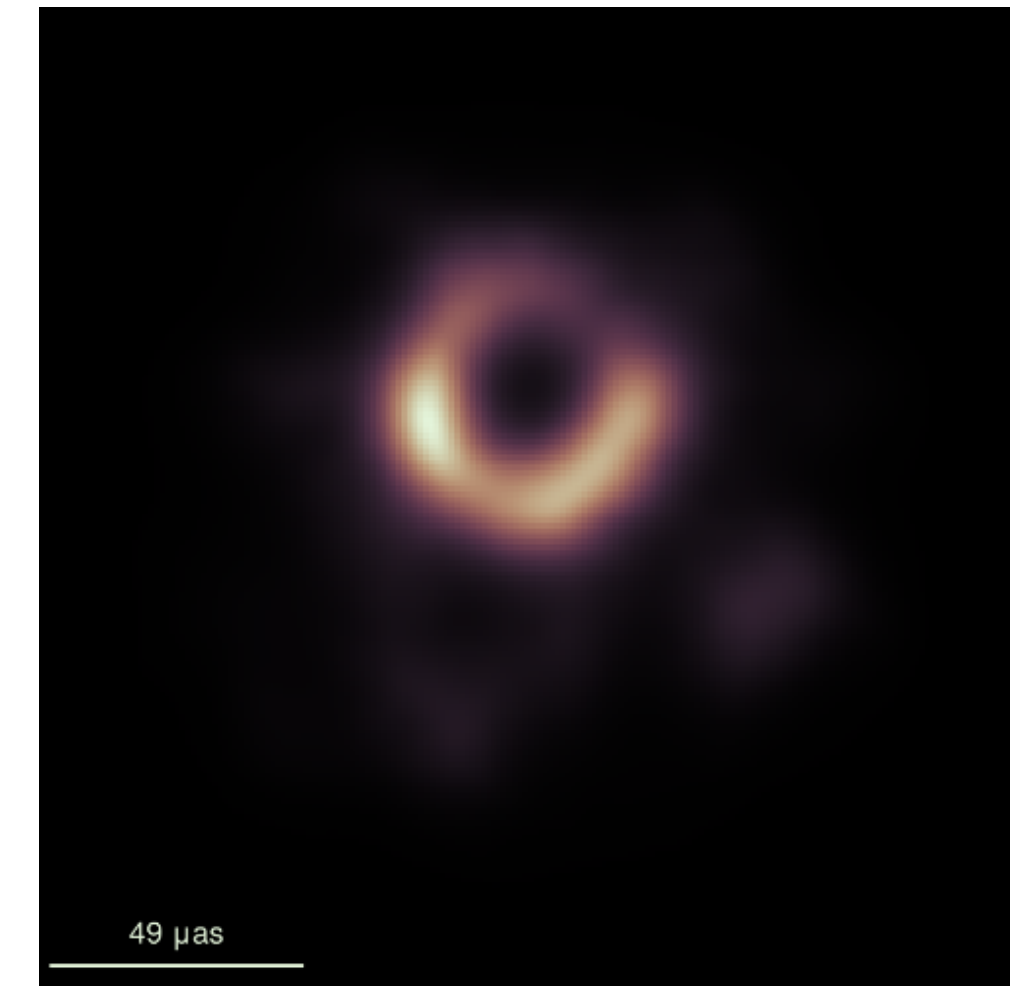
Reactant + Machine Learning



Reactant + Comrade

Made with Reactant + EnzymeMLIR

- End-to-end compilation of EHT imaging pipeline with Reactant, capable of reproducing original results
- Successful end-to-end execution on accelerators
- Preliminary analysis shows substantial improvements in speed for images with high resolution
- >15x for 2k images on a single consumer GPU (NVIDIA 3090 @ Float64)



Computing Hardware is No Longer For Everybody

Computing Hardware is No Longer For Everybody

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

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 December 6, 2022. REUTERS/Yves Herman/File 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

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 December 6, 2022. REUTERS/Yves Herman/File 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 Trainium2 and model distillation in Amazon Bedrock

Dec 3, 2024 • 3 min read

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 December 6, 2022. REUTERS/Yves Herman/File 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 Trainium2 and model distillation in Amazon Bedrock

Dec 3, 2024 • 3 min read

Elon Musk's xAI is reportedly trying to borrow \$12,000,000,000 for even 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.



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 December 6, 2022. REUTERS/Yves Herman/File 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 Tr

Dec 3, 2024 · 3 min read

OpenAI's Sam Altman is dreaming of running 100 million GPUs in the future - 100x more than it plans to run by December 2025

News

By [Efosa Udinmwun](#) published July 26, 2025

OpenAI scale-up will give its investors something to think about



Elon Musk's xAI is reportedly trying to borrow \$12,000,000,000 for even more Nvidia GPUs, an impulse all PC gamers can truly understand

News

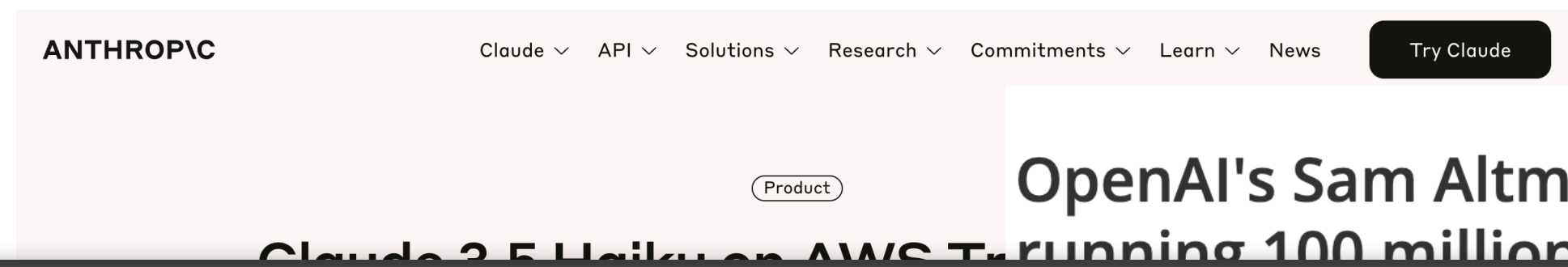
By [Andy Edser](#) published 23 July 2025

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



[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belg
Herman/File Photo Purchase Licensing Rights ↗

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

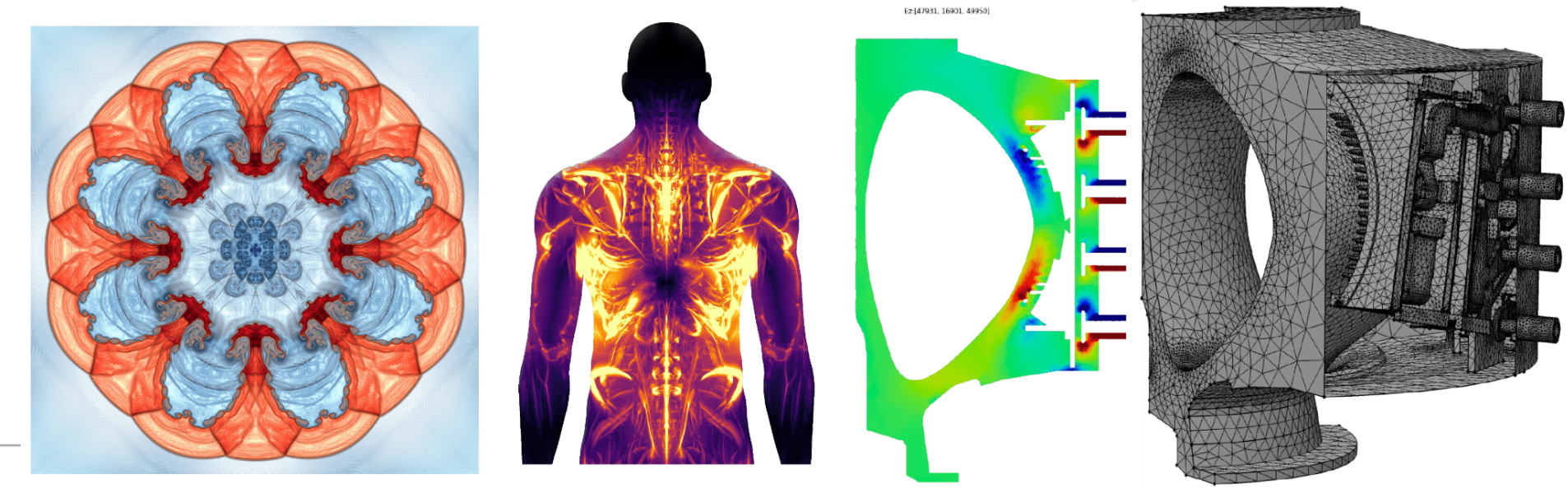


nothing to think about

Ironwood: The first Google TPU for the age of inference

 Comments (2)

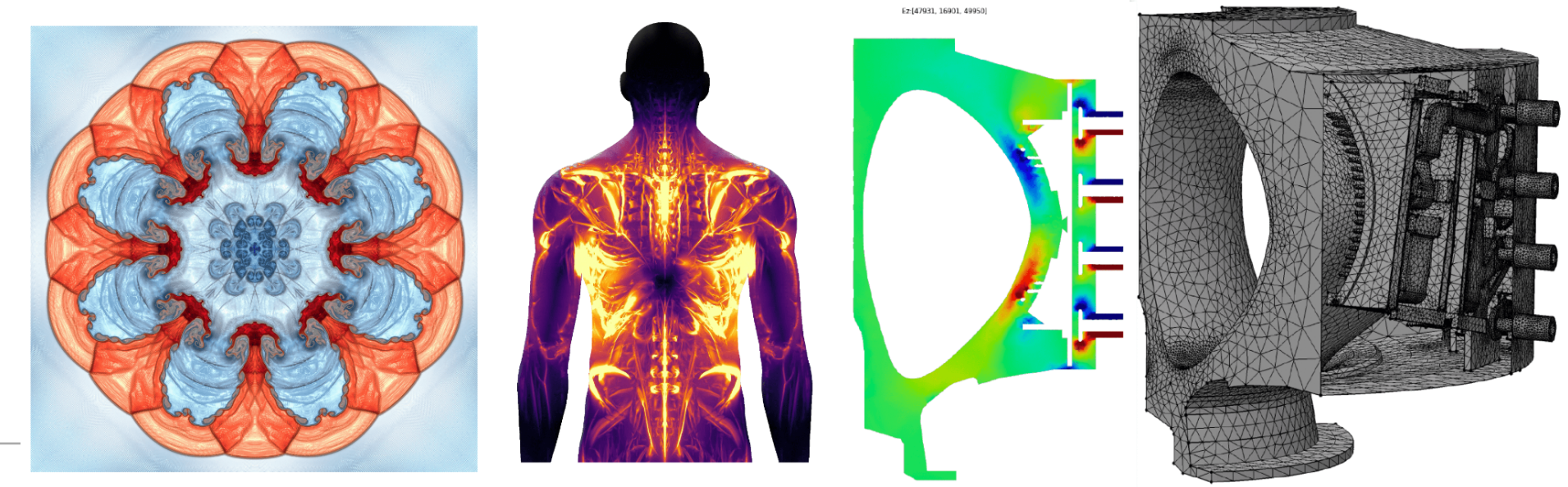
Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

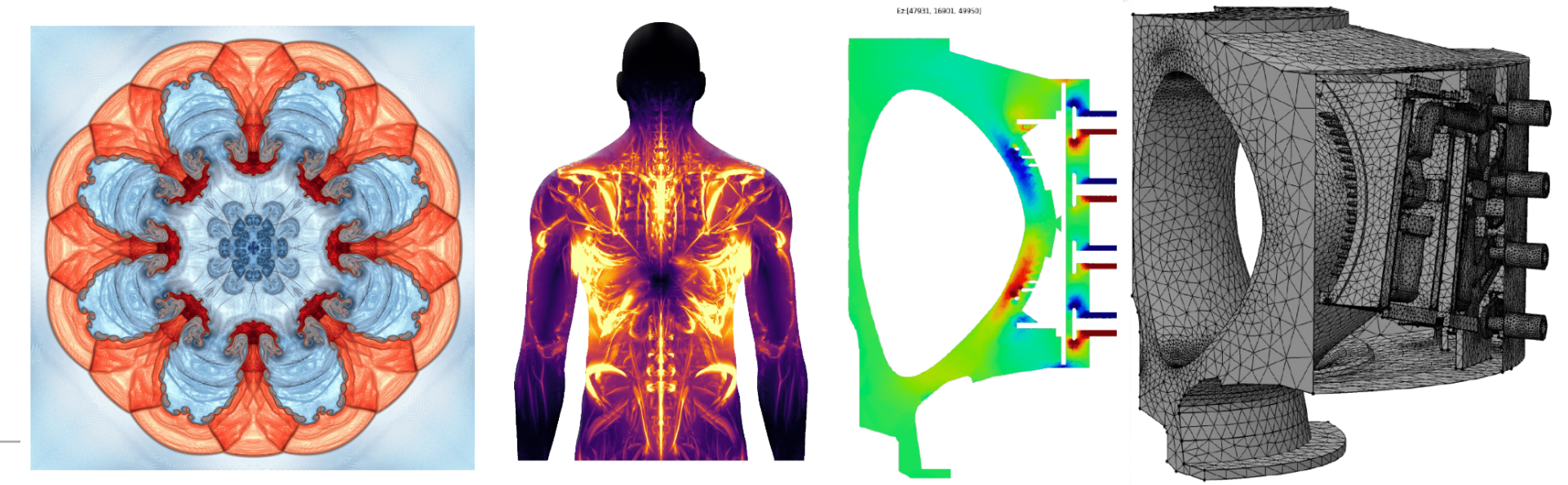
Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

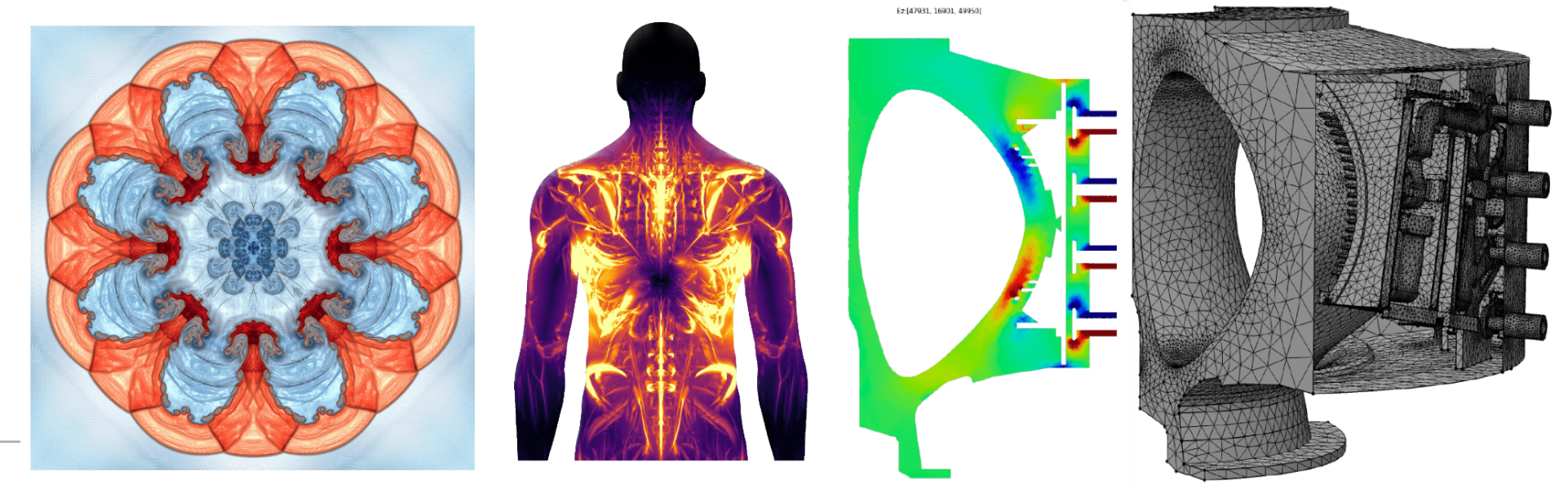

Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

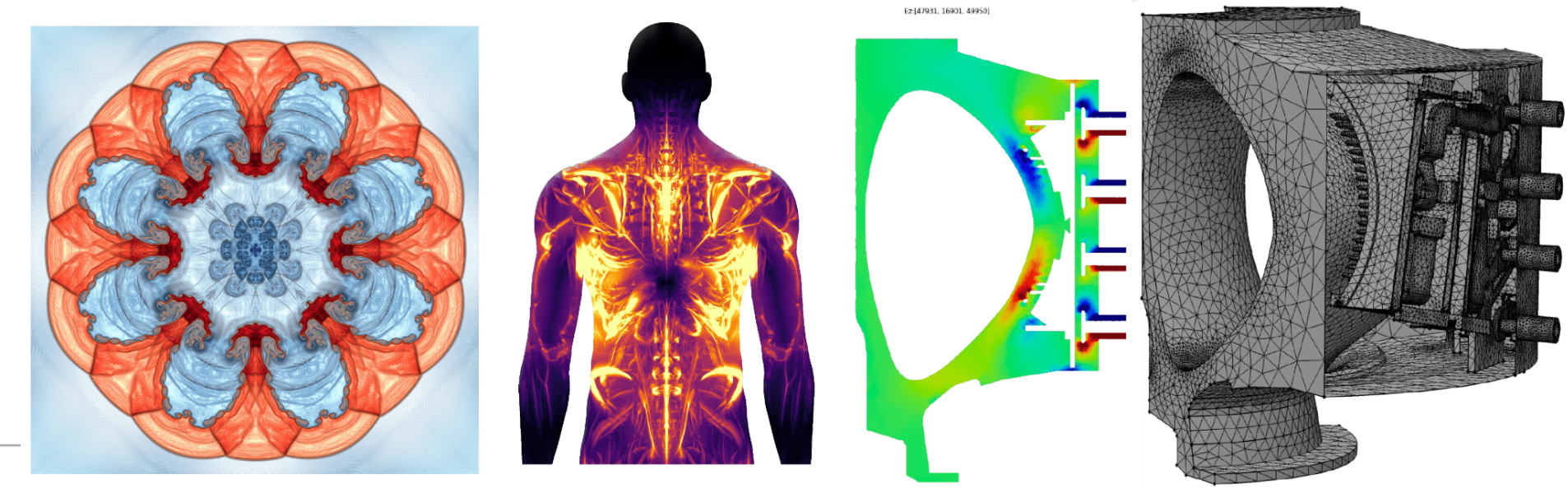
Lingua Franca of Scientific Computing



- 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);
    }
};
```

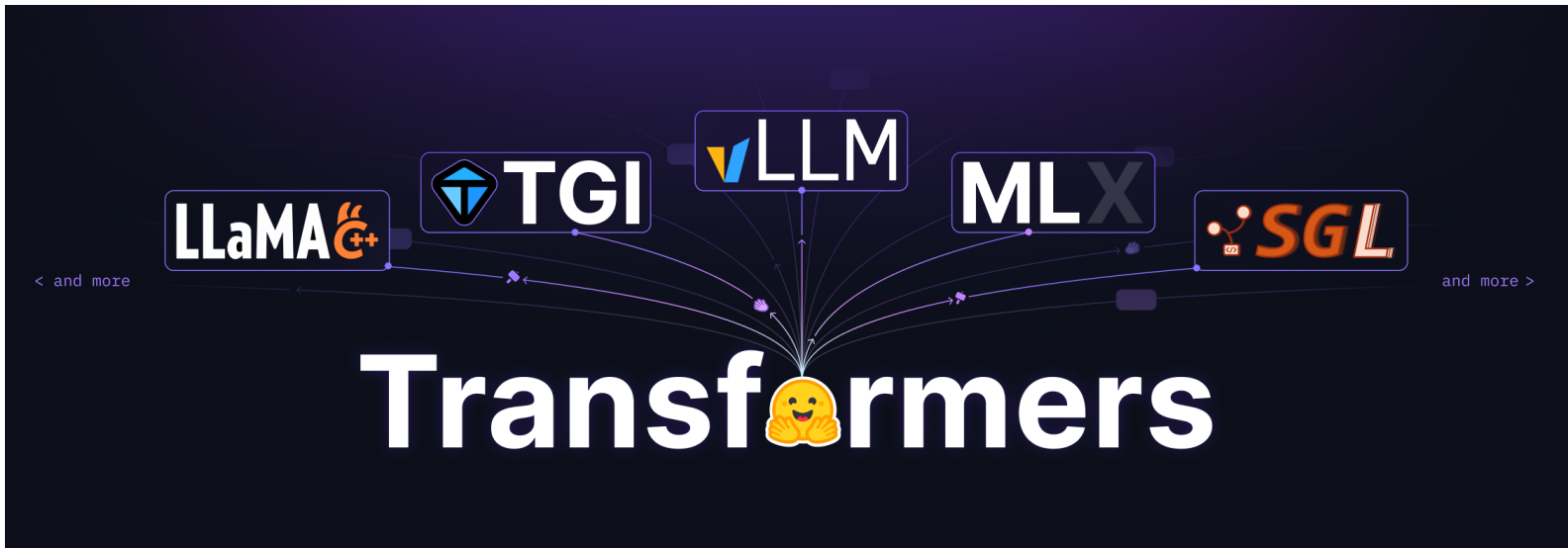
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                     Index_t padded_numNode,
                                     const Int_t* nodeElemCount,
                                     const Int_t* nodeElemStart,
                                     const 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;
    }
}
```

How do we write ML Accelerator code now?

How do we write ML Accelerator code now?



Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with [Stability AI](#) and [Runway](#) and builds upon our previous work:

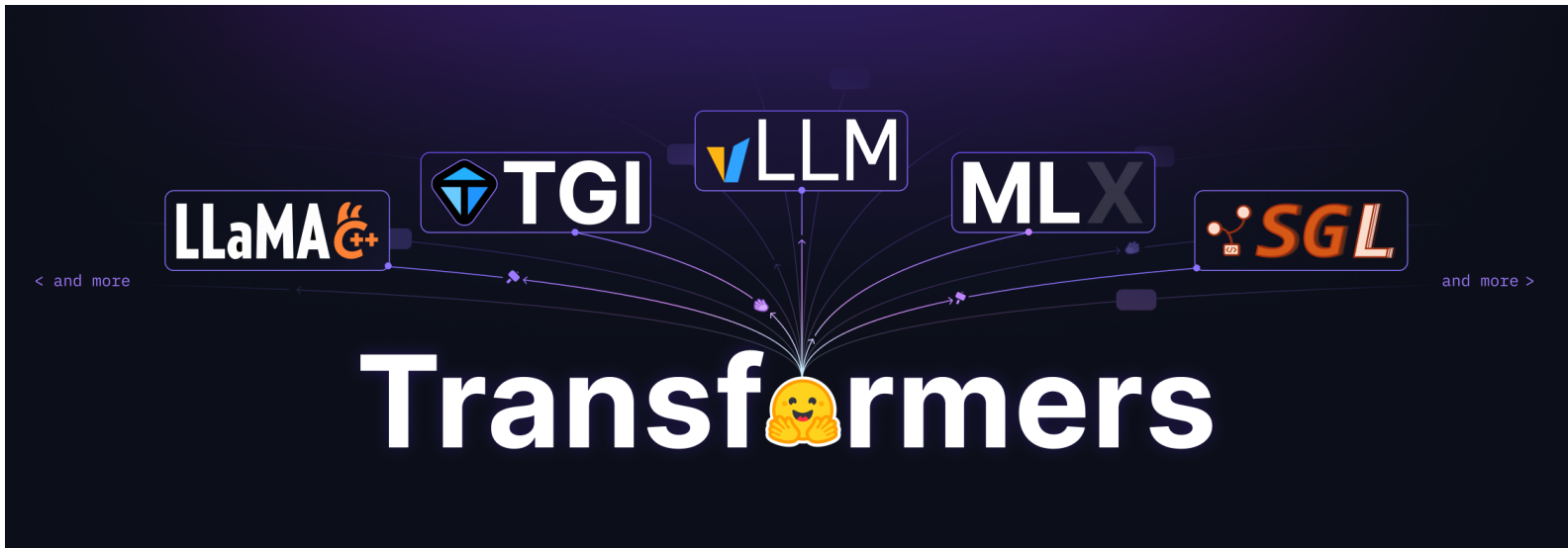
[High-Resolution Image Synthesis with Latent Diffusion Models](#)

[Robin Rombach*](#), [Andreas Blattmann*](#), [Dominik Lorenz](#), [Patrick Esser](#), [Björn Ommer](#)

[CVPR '22 Oral](#) | [GitHub](#) | [arXiv](#) | [Project page](#)



How do we write ML Accelerator code now?



Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with [Stability AI](#) and [Runway](#) and builds upon our previous work:

[High-Resolution Image Synthesis with Latent Diffusion Models](#)
[Robin Rombach*](#), [Andreas Blattmann*](#), [Dominik Lorenz](#), [Patrick Esser](#), [Björn Ommer](#)
[CVPR '22 Oral](#) | [GitHub](#) | [arXiv](#) | [Project page](#)



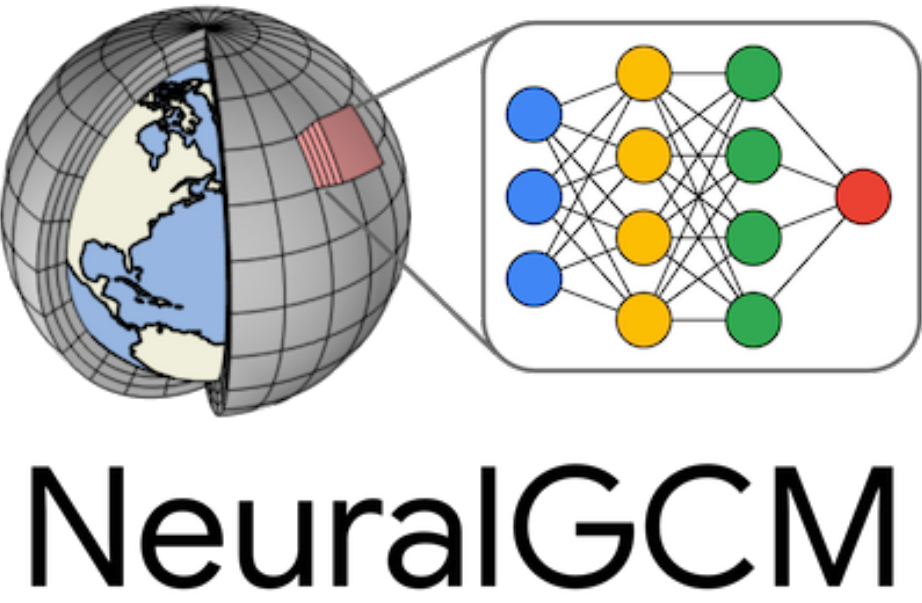
JAX, M.D.

Accelerated, Differentiable, Molecular Dynamics

[Quickstart](#) | [Reference docs](#) | [Paper](#) | [NeurIPS 2020](#)

 Build  passing DOI [10.5281/zenodo.14220247](#)  pypi  v0.2.8  license  Apache 2.0

Molecular dynamics is a workhorse of modern computational condensed matter physics. It is frequently used to simulate materials to observe how small scale interactions can give rise to complex large-scale phenomenology. Most molecular dynamics packages (e.g. HOOMD Blue or LAMMPS) are complicated, specialized pieces of code





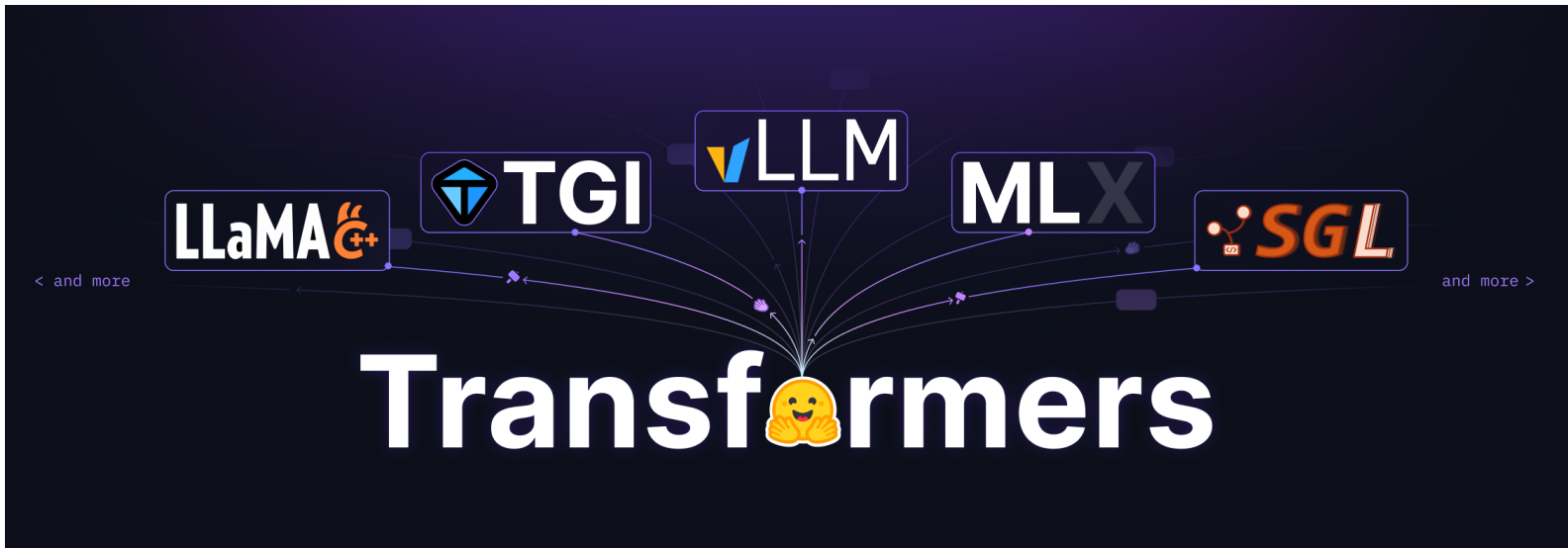
jaxspec

 PYPY  V0.3.0  PYTHON  >=3.10,<3.13  DOCS  PASSING  COVERAGE  94%  SLACK

⚠️ jaxspec is still in early release: expect bugs, breaking API changes, undocumented features and lack of functionalities

jaxspec is an X-ray spectral fitting library built in pure Python. It can currently load an X-ray spectrum (in the OGIP standard), define a spectral model from the implemented components, and calculate the best parameters using state-of-the-art Bayesian approaches. It is built on top of JAX to provide just-in-time compilation and automatic differentiation of the spectral models, enabling the use of sampling algorithm such as NUTS.

How do we write ML Accelerator code now?



Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with [Stability AI](#) and [Runway](#) and builds upon our previous work:

[High-Resolution Image Synthesis with Latent Diffusion Models](#)
[Robin Rombach*](#), [Andreas Blattmann*](#), [Dominik Lorenz](#), [Patrick Esser](#), [Björn Ommer](#)
[CVPR '22 Oral](#) | [GitHub](#) | [arXiv](#) | [Project page](#)



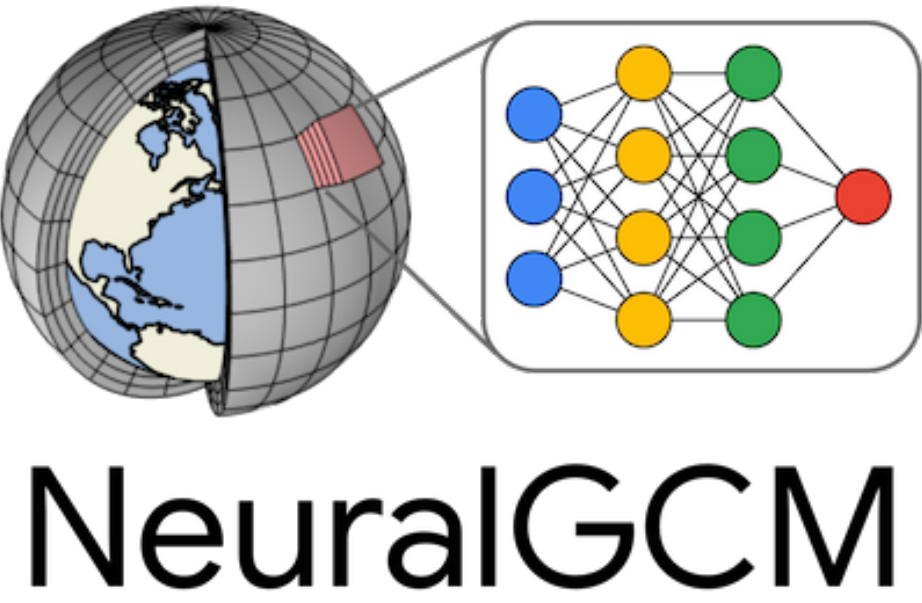
JAX, M.D.

Accelerated, Differentiable, Molecular Dynamics

[Quickstart](#) | [Reference docs](#) | [Paper](#) | [NeurIPS 2020](#)

Build passing DOI [10.5281/zenodo.14220247](#) pypi v0.2.8 license Apache 2.0

Molecular dynamics is a workhorse of modern computational condensed matter physics. It is frequently used to simulate materials to observe how small scale interactions can give rise to complex large-scale phenomenology. Most molecular dynamics packages (e.g. HOOMD Blue or LAMMPS) are complicated, specialized pieces of code





jaxspec

PYPI v0.3.0 PYTHON >=3.10,<3.13 DOCS PASSING COVERAGE 94% SLACK

⚠ jaxspec is still in early release: expect bugs, breaking API changes, undocumented features and lack of functionalities

jaxspec is an X-ray spectral fitting library built in pure Python. It can currently load an X-ray spectrum (in the OGIP standard), define a spectral model from the implemented components, and calculate the best parameters using state-of-the-art Bayesian approaches. It is built on top of JAX to provide just-in-time compilation and automatic differentiation of the spectral models, enabling the use of sampling algorithm such as NUTS.

Rewrite it in JAX/PyTorch!

The Exascale Computing Project (ECP) ECP by the Numbers

The ECP ran from 2016–2024 and was the largest software research, development, and deployment project managed to date by the US Department of Energy (DOE). The \$1.8 billion project was a joint effort by the DOE Office of Science and the National Nuclear Security Administration that funded nearly 2,800 multidisciplinary individuals over the lifetime of the project to uplift the high-performance computing community toward capable exascale platforms, software, and application codes. The outcome was the delivery of an exascale computing ecosystem to provide breakthrough solutions that address future challenges in energy assurance, economic competitiveness, healthcare, and scientific discovery, as well as growing security threats. The ECP exascale ecosystem includes DOE mission-critical application codes, the underlying supporting software technologies, and mechanisms for their deployment and integration.

ECP was a grand convergence of advances in modeling and simulation, software tools and libraries, data analytics, machine learning, and artificial intelligence in support of delivering the world's first capable exascale ecosystem.

The payoff is here: exascale computing is revolutionizing nearly every domain of science.

Created to develop the nation's first capable exascale computing ecosystem, this unprecedented DOE research, development, and deployment project has already made a huge impact on computational science:



2,800 collaborators funded to develop exascale applications, software, and hardware.



Game-changing results in a broad spectrum of science and engineering application areas.



2 different GPU architectures now proven to work with exascale environments.



First and only open-source scientific software stack developed for scalability and available across all HPC platforms, including cloud computing.

The Exascale Computing Project (ECP) ECP by the Numbers

The ECP ran from **2016–2024** and was the largest software research, development, and deployment project managed to date by the US Department of Energy (DOE). The **\$1.8 billion** project was a joint effort by the DOE Office of Science and the National Nuclear Security Administration that funded nearly 2,800 multidisciplinary individuals over the lifetime of the project to uplift the high-performance computing community toward capable exascale platforms, software, and application codes. The outcome was the delivery of an exascale computing ecosystem to provide breakthrough solutions that address future challenges in energy assurance, economic competitiveness, healthcare, and scientific discovery, as well as growing security threats. The ECP exascale ecosystem includes DOE mission-critical application codes, the underlying supporting software technologies, and mechanisms for their deployment and integration.

ECP was a grand convergence of advances in modeling and simulation, software tools and libraries, data analytics, machine learning, and artificial intelligence in support of delivering the world's first capable exascale ecosystem.

The payoff is here: exascale computing is revolutionizing nearly every domain of science.

Created to develop the nation's first capable exascale computing ecosystem, this unprecedented DOE research, development, and deployment project has already made a huge impact on computational science:



2,800 collaborators funded to develop exascale applications, software, and hardware.



Game-changing results in a broad spectrum of science and engineering application areas.



2 different GPU architectures now proven to work with exascale environments.



First and only open-source scientific software stack developed for scalability and available across all HPC platforms, including cloud computing.

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**

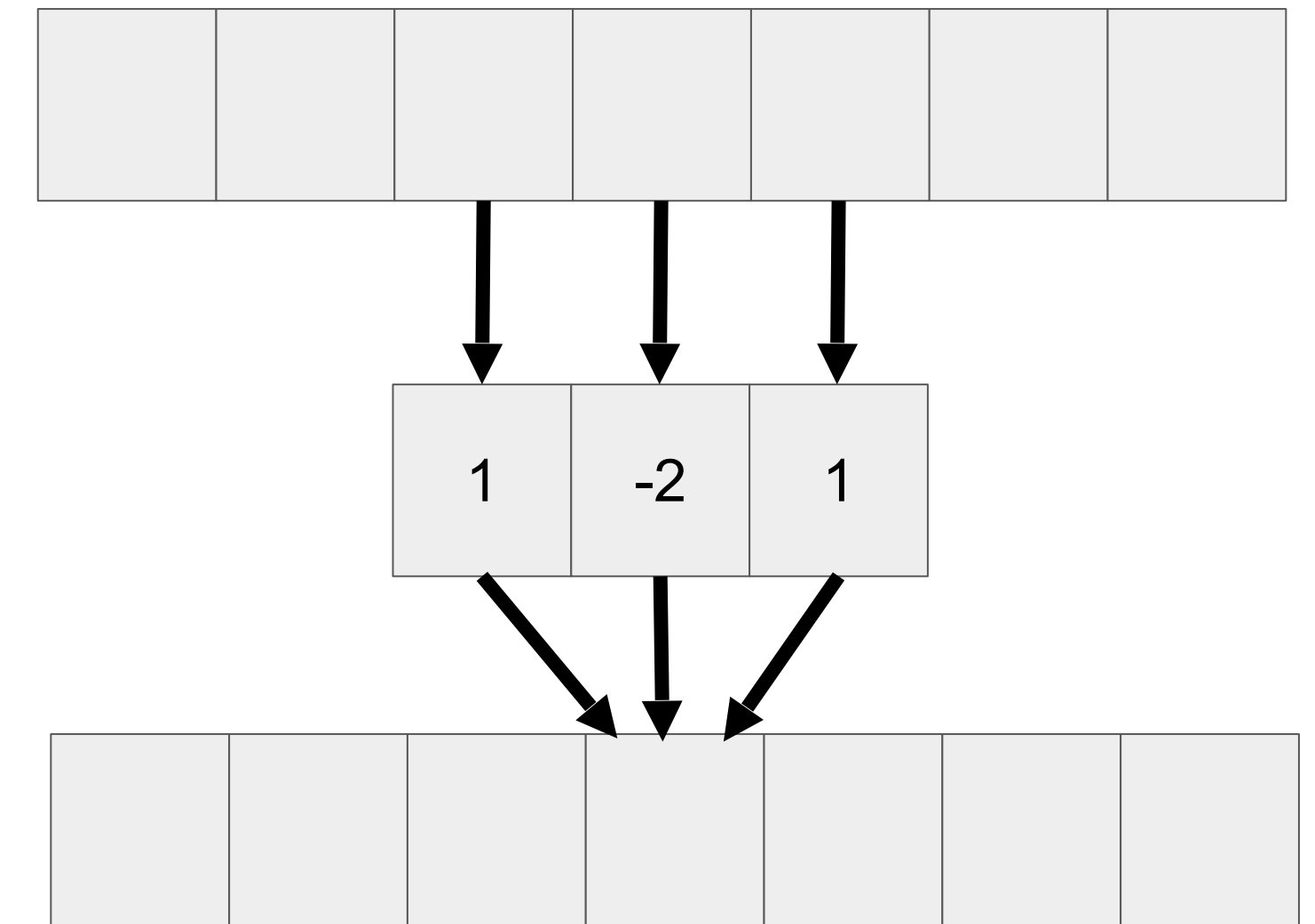
Buildkite CPU+GPU **failing** Docker **invalid**

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 {
  ...
}
```

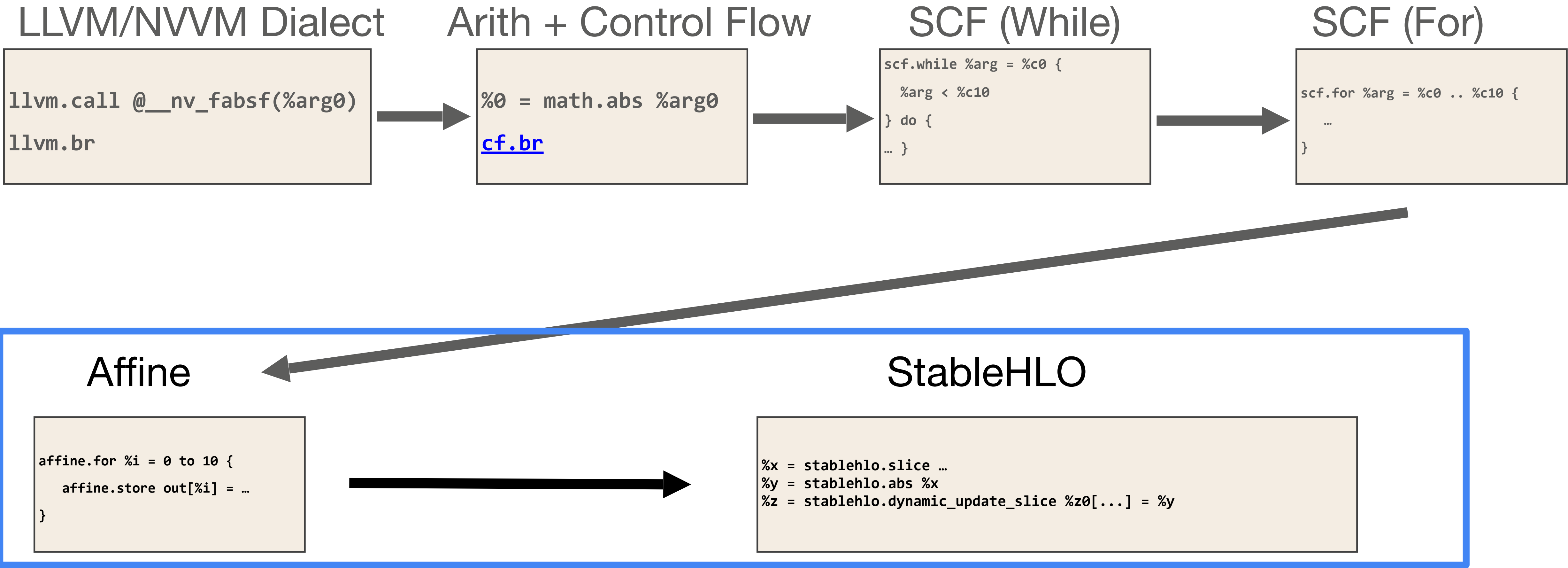
Affine

```
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



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 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 lost the convolution!

- Fewer local optimizations
- Slower and
- Less structured

```
%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

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 3.45% over XLA’s optimization flow across our benchmark suite on various CPU and GPU platforms, with a maximum speedup of 56.26% for NasRNN 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/nnnnnnn.nnnnnnn>

*These authors contributed equally.

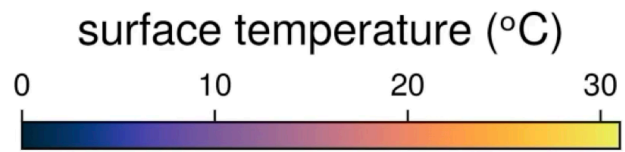
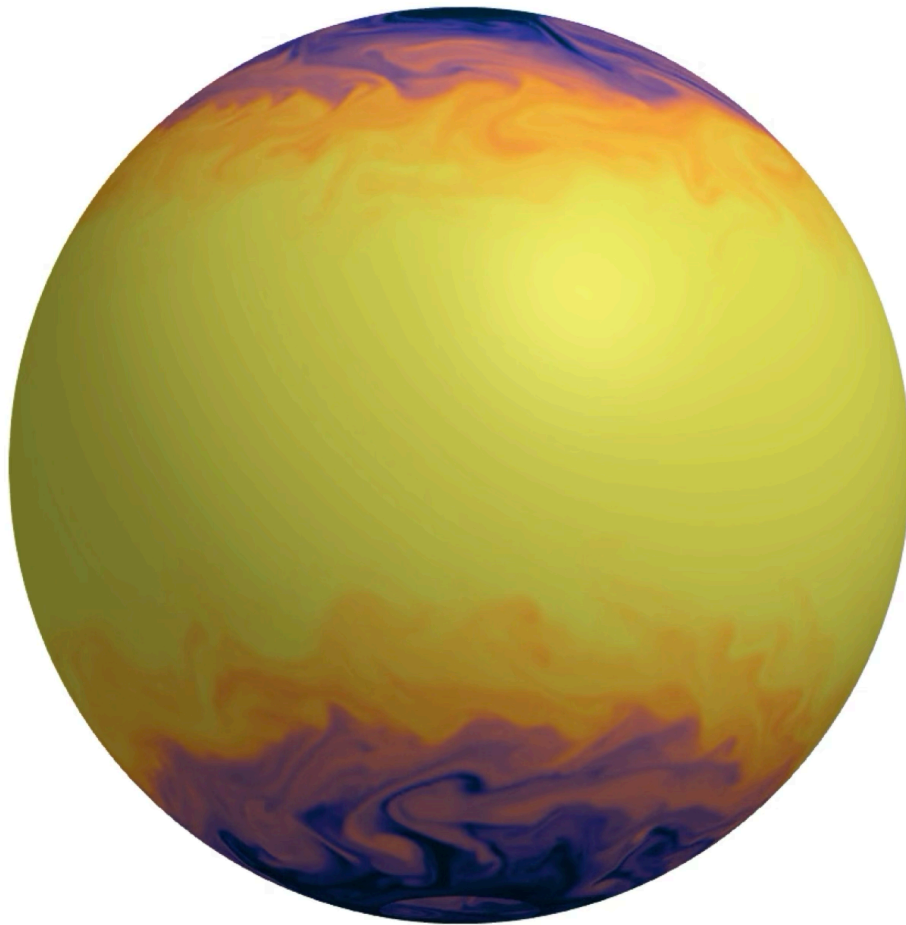
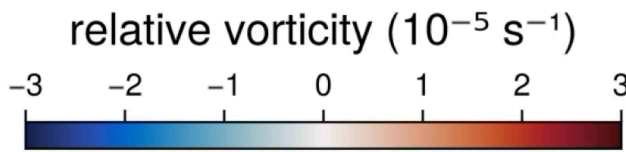
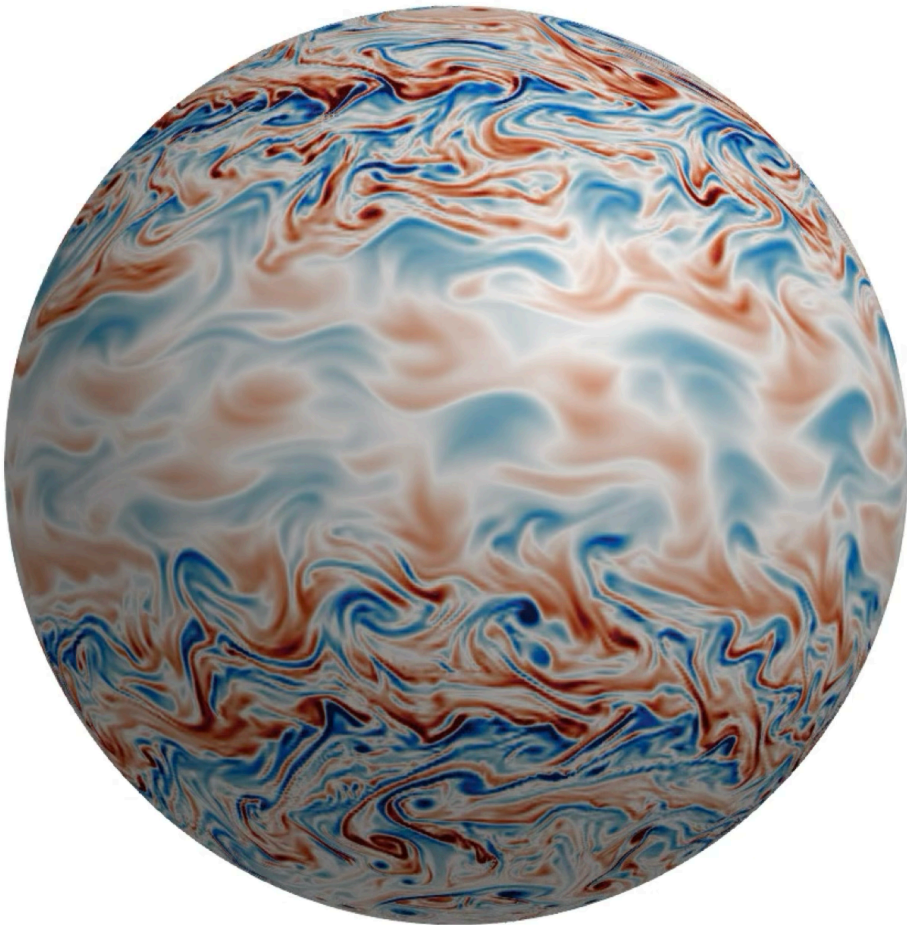
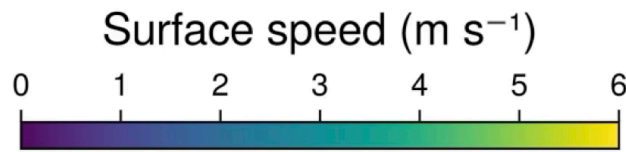
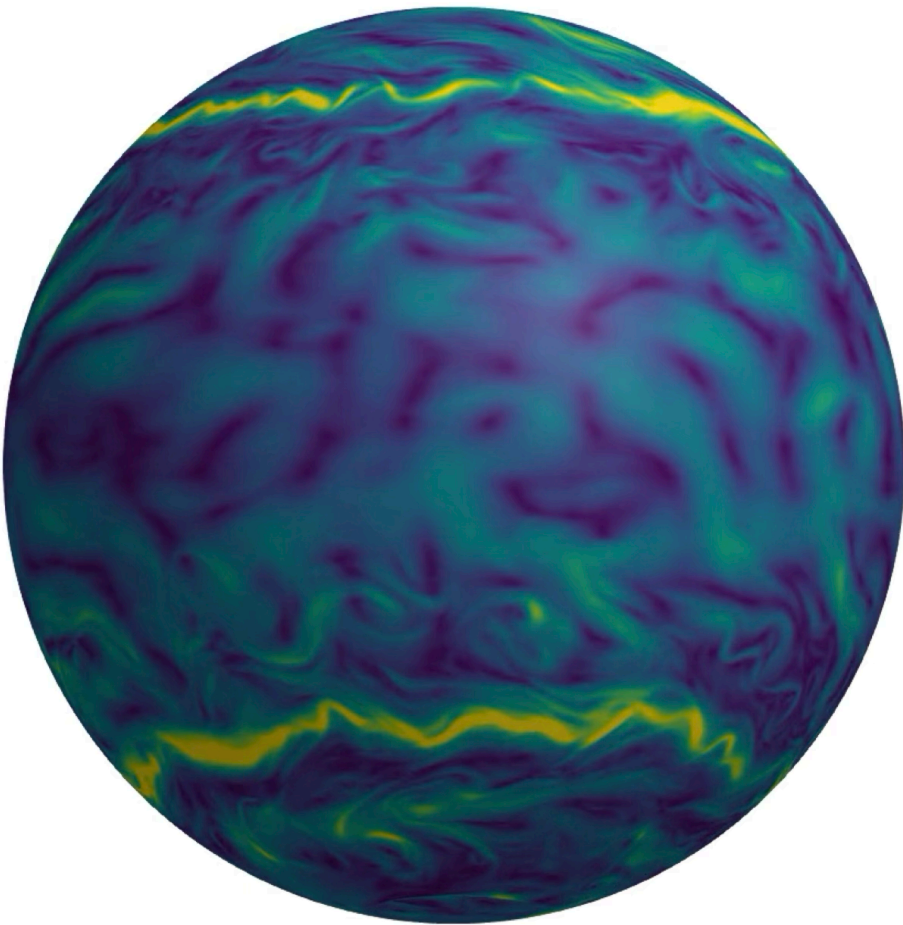
Authors’ addresses: Arya Vohra, aryavohra@uchicago.edu, University of Chicago, USA; Leo Seojun Lee, seojun.lee@oriel.ox.ac.uk, University of Oxford, UK; Jakub Bachurski, kbachurski@gmail.com, University of Cambridge, UK; Oleksandr Zinenko, alex@brium.ai, Brium, France; Phitchaya Mangpo Phothilimthana,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

CUDA to Accelerator IR (StableHLO)

165.0 days



```
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]>
```

Single Node Code => Multi Node Code

- Each tensor and operation was automatically sharded across the cluster
- Creates suboptimal shardings for intermediate values

```
%left = stablehlo.slice %x[1:2] : tensor<100x100> -> tensor<1x100>  
%x1 = stablehlo.dynamic_update_slice %x[0:1] = %left : tensor<100x100>
```

- Creates redundant communications

```
%left1 = enzymexla.rotate_left %x, 1  
%left2 = enzymexla.rotate_left %x, 2  
%left3 = enzymexla.rotate_left %x, 3  
%result = %left1 + %left2 + %left3
```

- Built novel distributed operations and optimizations

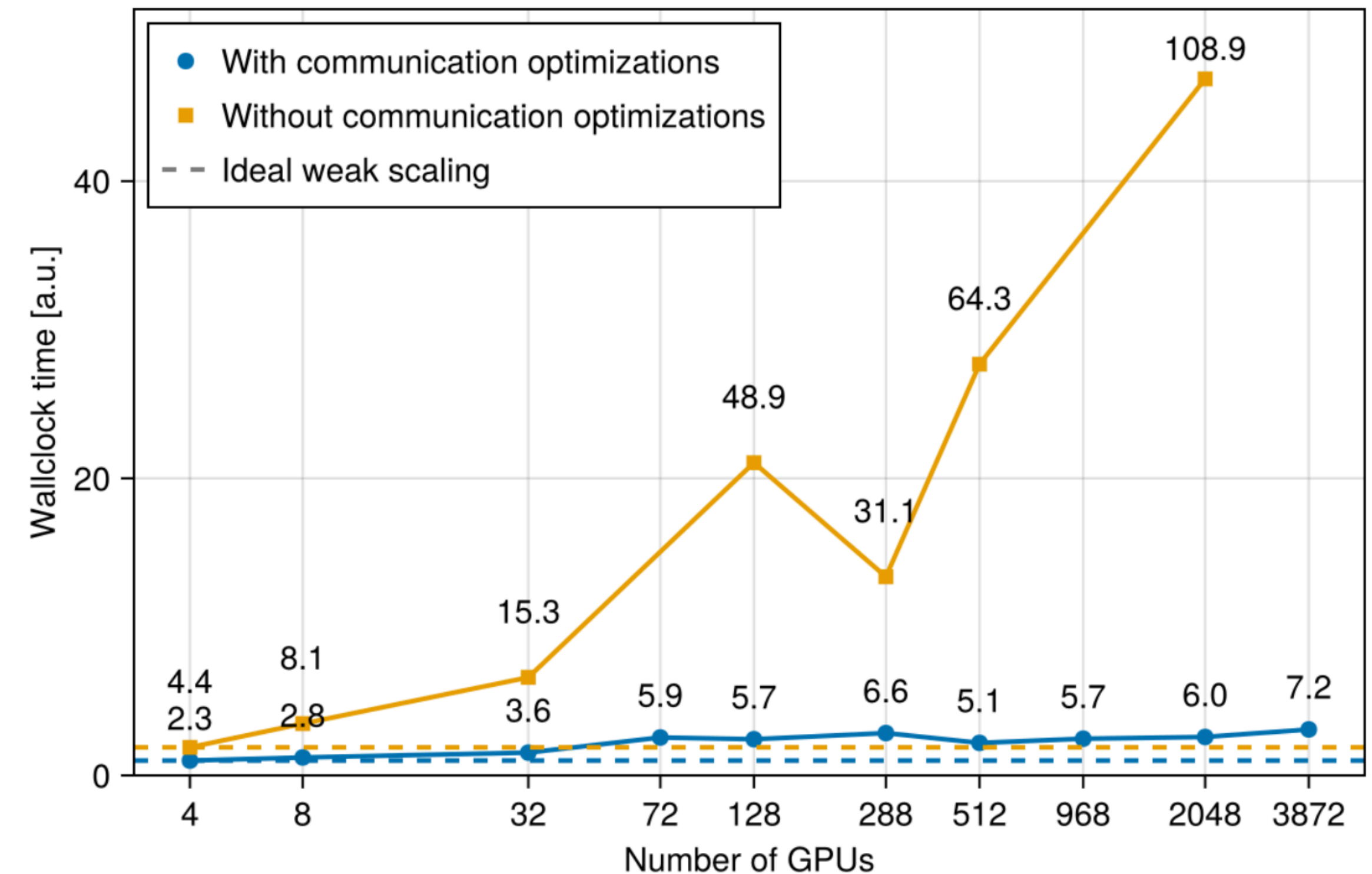
Distributed Optimizations

- Remove redundant communications (akin to automatic discovery of manual halo exchange)

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

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

- Fuse resharding operations into consumers to avoid unnecessary sends
- Follow sub tensor provenance & create novel communication partial-CSE to avoid (partially) redundant sends



Accelerator-Specific Optimizations

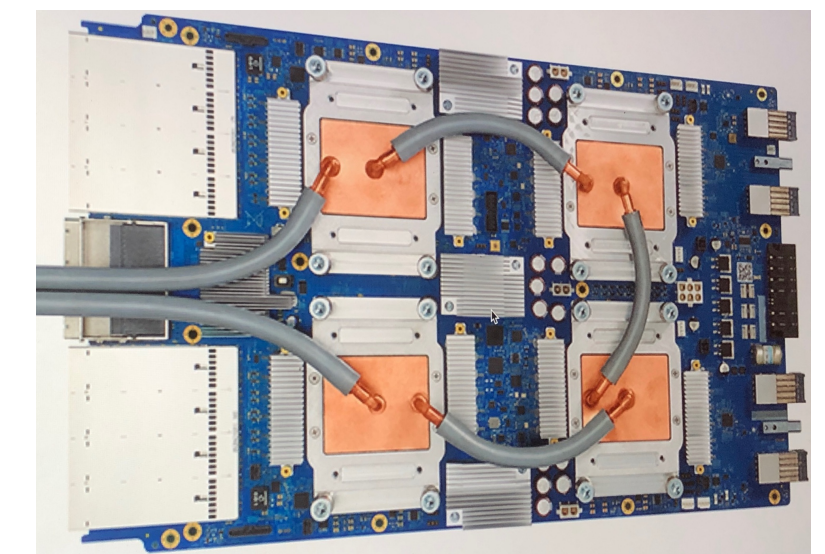
- ML accelerators contain the majority of their flops in specialized matrix cores. Detection of convolutions are useful, but stencils contain more general patterns than a pure stencil
 - $2a - b \Rightarrow [2, -1]^T * [a, b]$
 - Generalization of Im2Col
- Accelerators often don't contain native f64 (or even f32) operations.
 - Automatic rewriting of higher level floats in terms of multiple limbs of smaller floats to preserve accuracy while maintaining performance.
 - Even with the additional work, converting f32->2x bf16 was 10x faster than f32 due to bf16-specific hardware

Performance Results

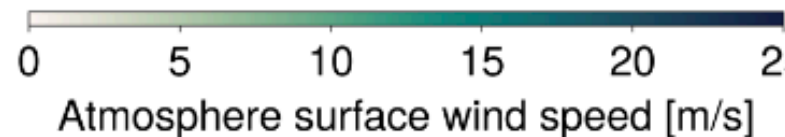
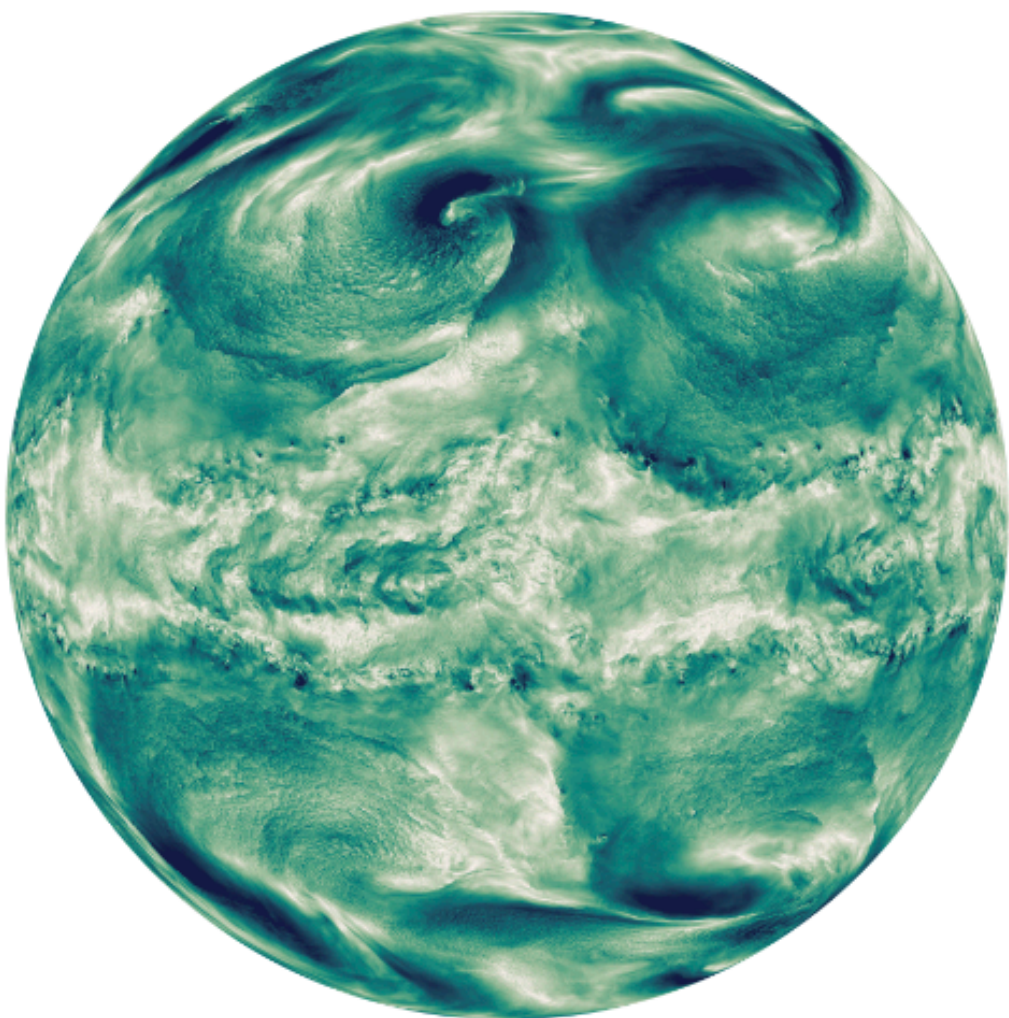
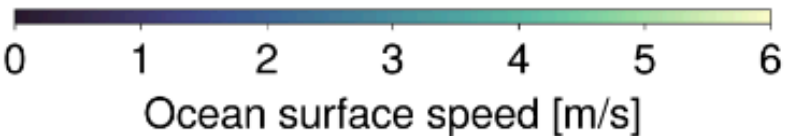
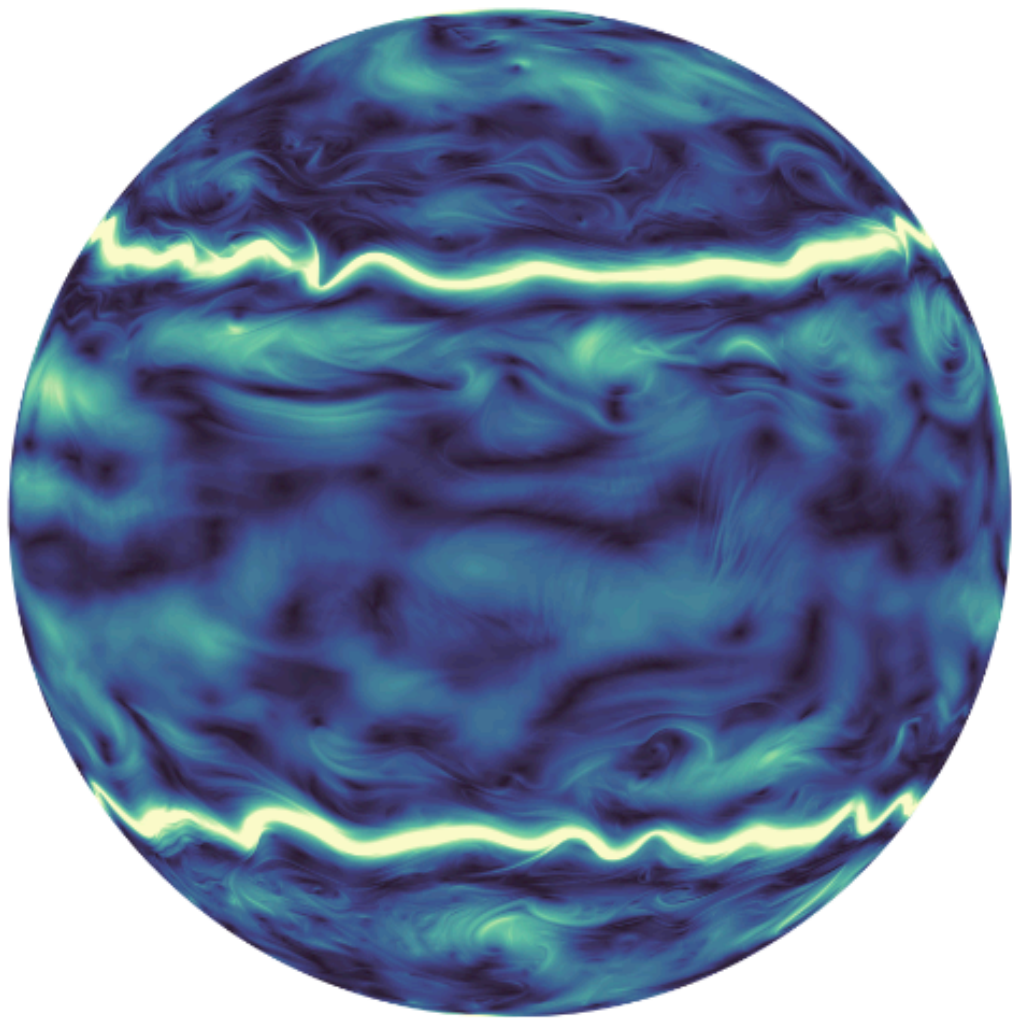
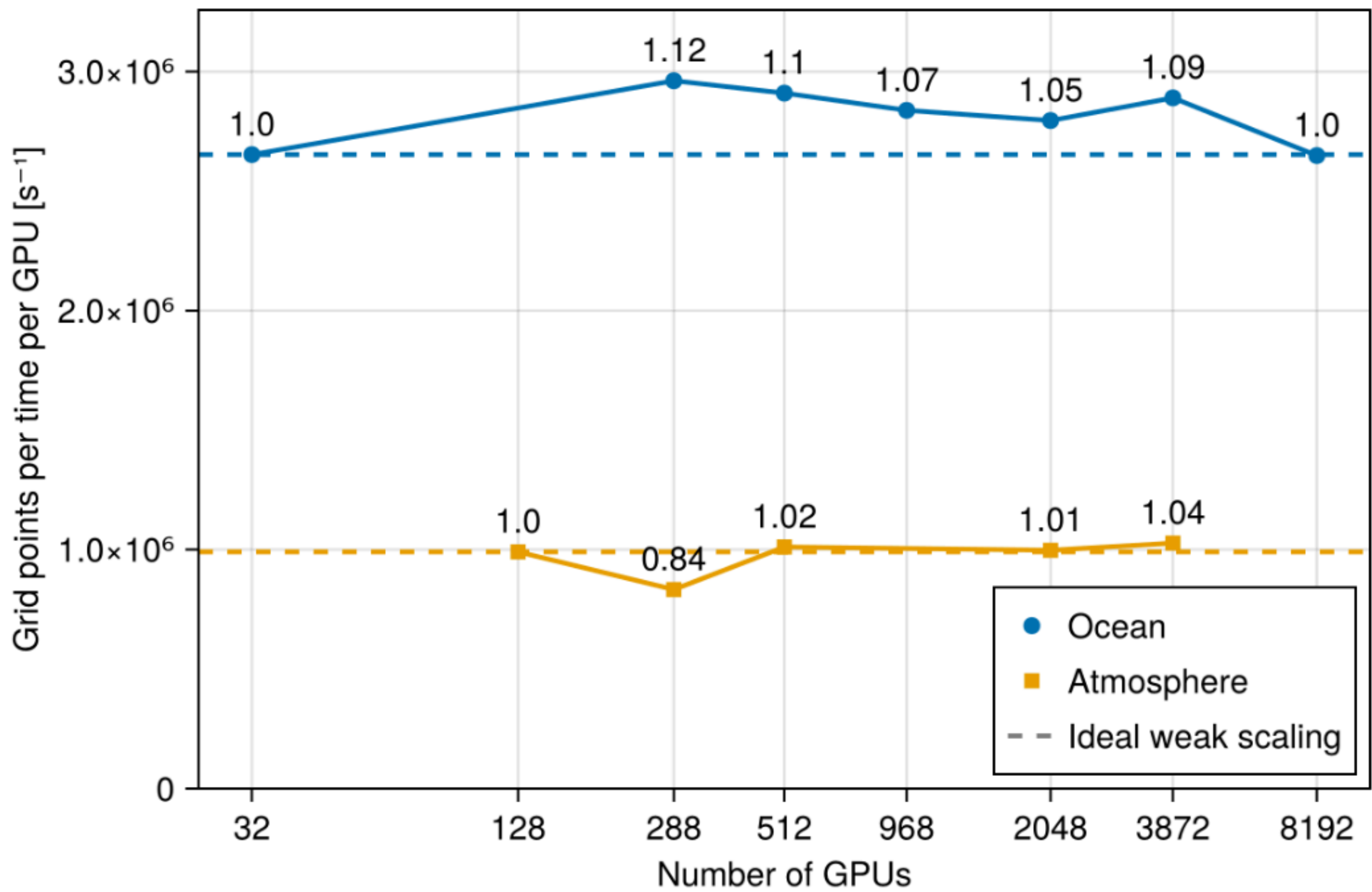
- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
 - Alps (2688 nodes x 4 NVIDIA GH200 GPUs)
 - Perlmutter (1536 nodes x 4 NVIDIA A100 GPUs)
 - 8,192 Google TPUs v7 (4614 F8 TFLOPS each)
- Good Single-Node Perf (CPU)
 - Vanilla Model: 272.0seconds
 - Tensor Optims: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



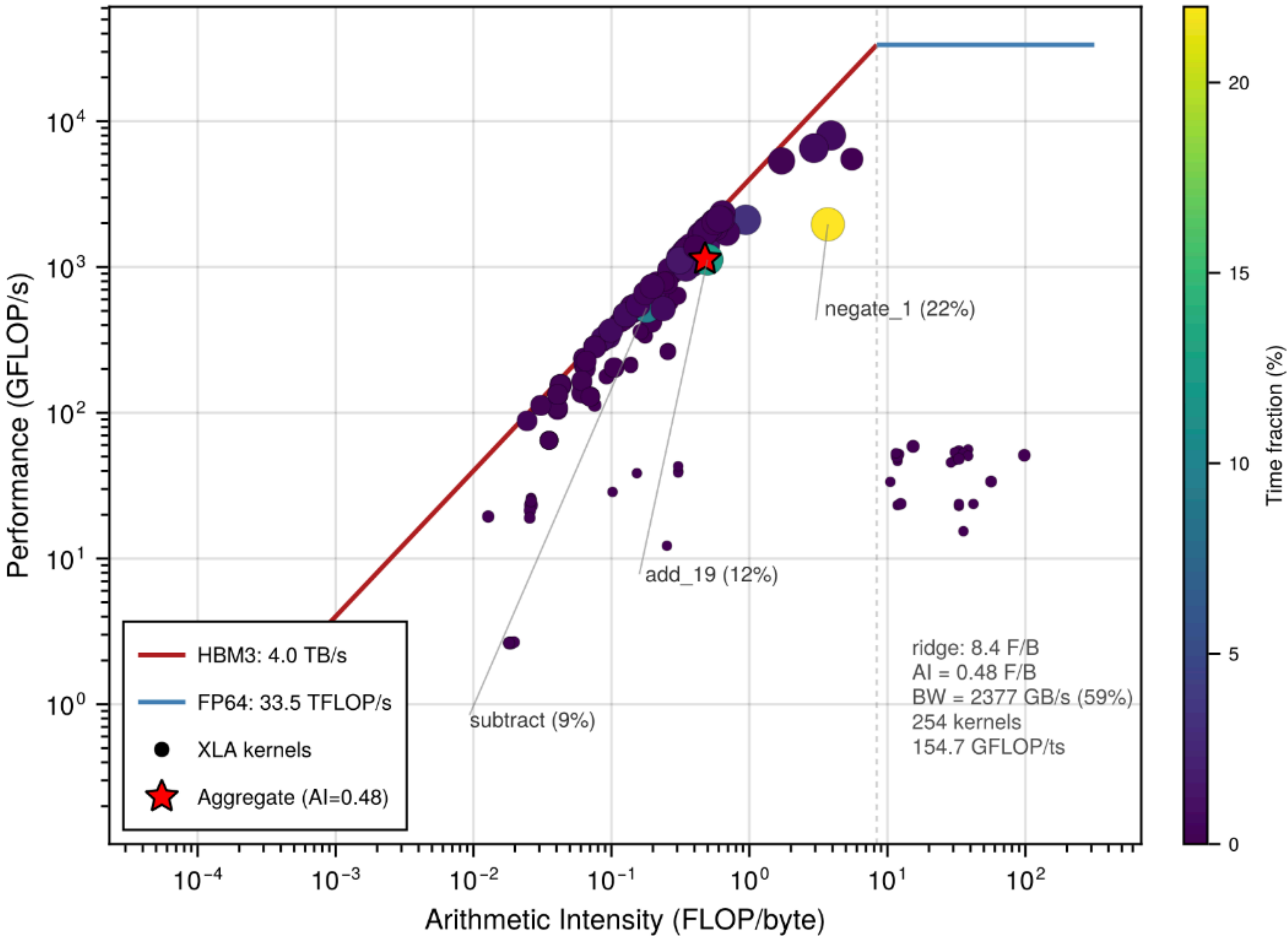
Google Cloud



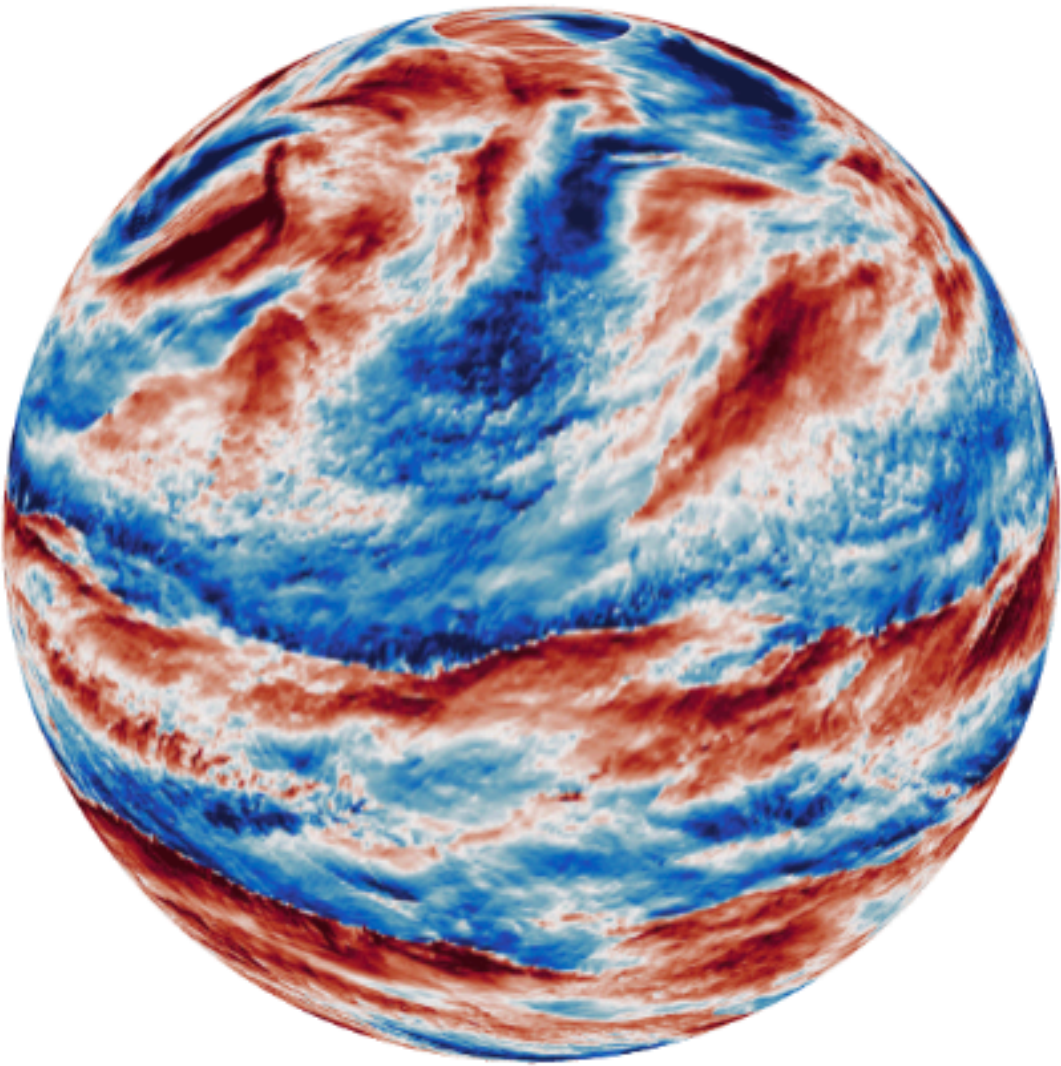
Performance Results (Alps / NVIDIA GPU)



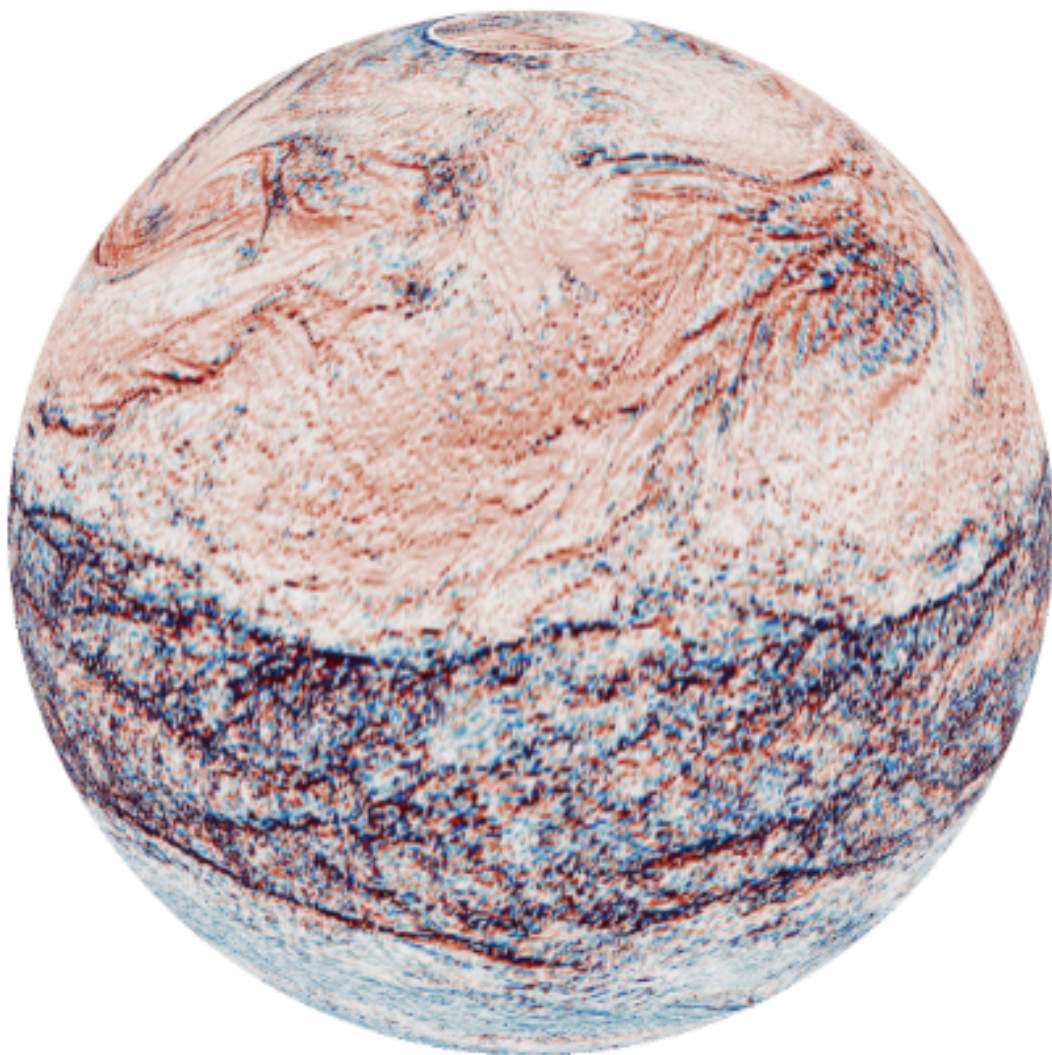
Performance Results (Alps / NVIDIA GPU)



Meridional momentum, ρv

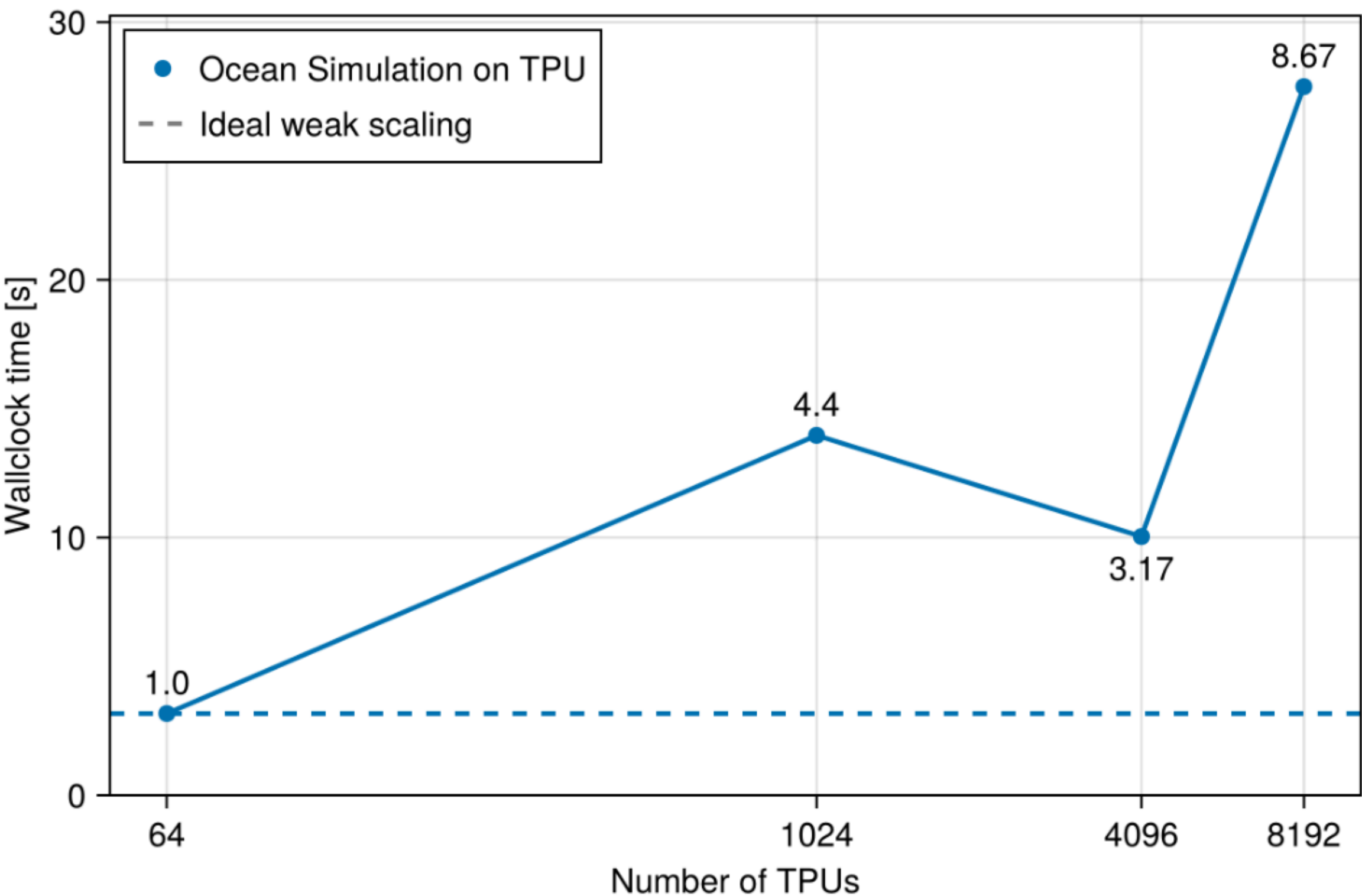


Sensitivity $\partial J / \partial(\rho v_0)$
 $J = V^{-1} \int (\rho \theta)^2 dV$



Performance Results (Google Cloud / TPUs)

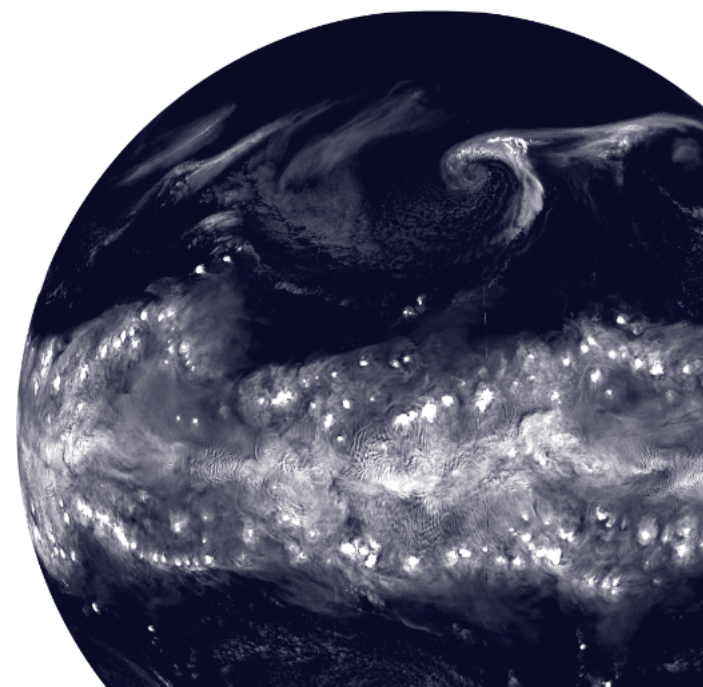
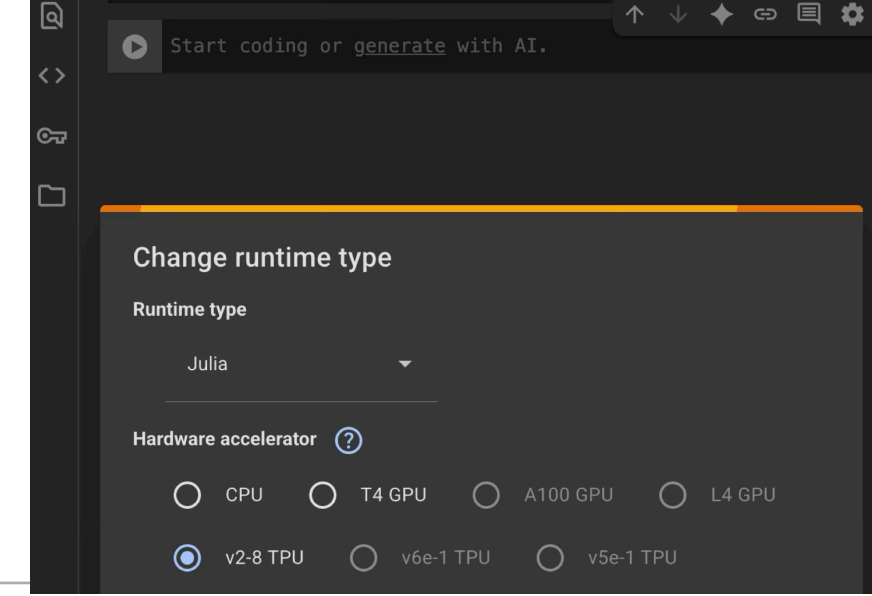
- At peak utilized of 97% of the TPU cluster, containing 36 FP8 exaFLOP!
- 1.493 PiB of HBM to store fields!
- 8x128 registers created some additional comms existing patterns don't yet handle (in progress)



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%

Conclusions

- Reactant: Optimizing Compiler Framework For Julia, built on top of MLIR
 - Preserves and optimizes high-level structures from Julia code into the compiler
 - Compatible with mutation, control flow(*), autodiff (Enzyme)
- Computing hardware is increasingly moving to domain-specific accelerators, leaving existing scientific workloads in the dust
- Reactant extracts accelerator-friendly tensor operators from existing parallel code and run them on distributed accelerators (incl CPU/GPU/TPU/Trainium)
- Opens the door for moving workloads to where you want to run them, without needing to re-engineer them
- Works on generic LLVM, with explicit frontends for C++ and Julia
github.com/EnzymeAD/Reactant.jl and github.com/EnzymeAD/Reactant



Reactant.jl

- Optimizing Compiler Framework For Julia, built on top of MLIR
- Preserves high-level structures from Julia code into the compiler
 - Effectively optimize programs
 - Effectively retarget programs (allowing them to run on distributed clusters of your favorite accelerator, including CPU/GPU/TPU)
- Can leverage XLA (state of the art runtime, used by TensorFlow, JaX, & PyTorch)
- Compatible with mutation, control flow(*), autodiff (Enzyme)
- All open source ([GitHub.com/EnzymeAD/Reactant.jl](https://github.com/EnzymeAD/Reactant.jl))



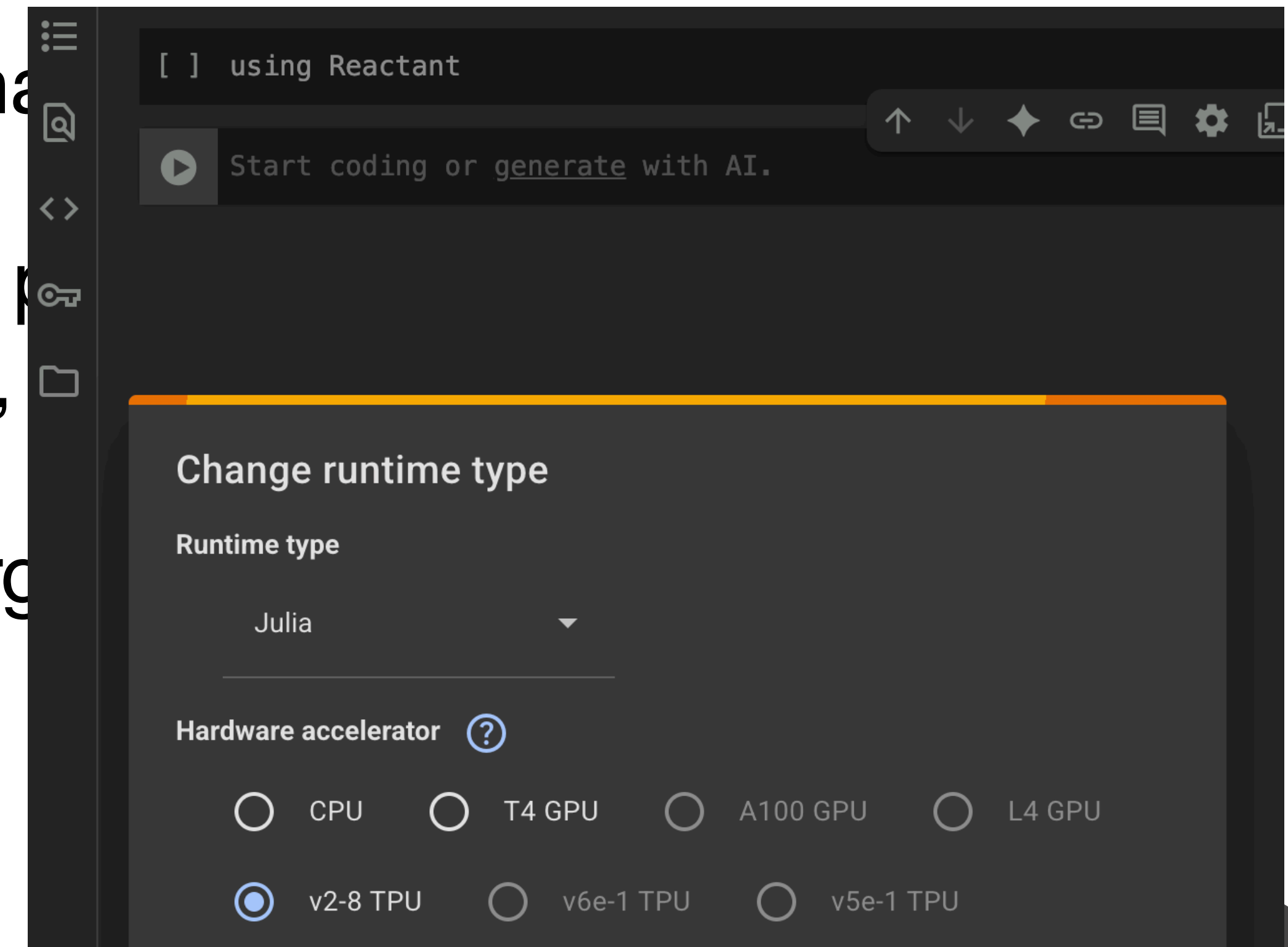
Reactant.jl Backend and Communication

- Reactant enables you to write regular Julia code which will automatically be distributed across different accelerators
 - Under the hood, Reactant recognizes what devices you have available
 - Generate corresponding communication primitives (e.g. MPI/NCCL/etc), kernels, and library calls (BLAS, cuBLAS, tpuBLAS) to execute
- No need to modify your application to retarget different systems



Reactant.jl Backend and Communication

- Reactant enables you to write regular Julia code which will automatically be distributed across different accelerators
- Under the hood, Reactant recognizes what
- Generate corresponding communication kernels, and library calls (BLAS, cuBLAS,
- No need to modify your application to retarget

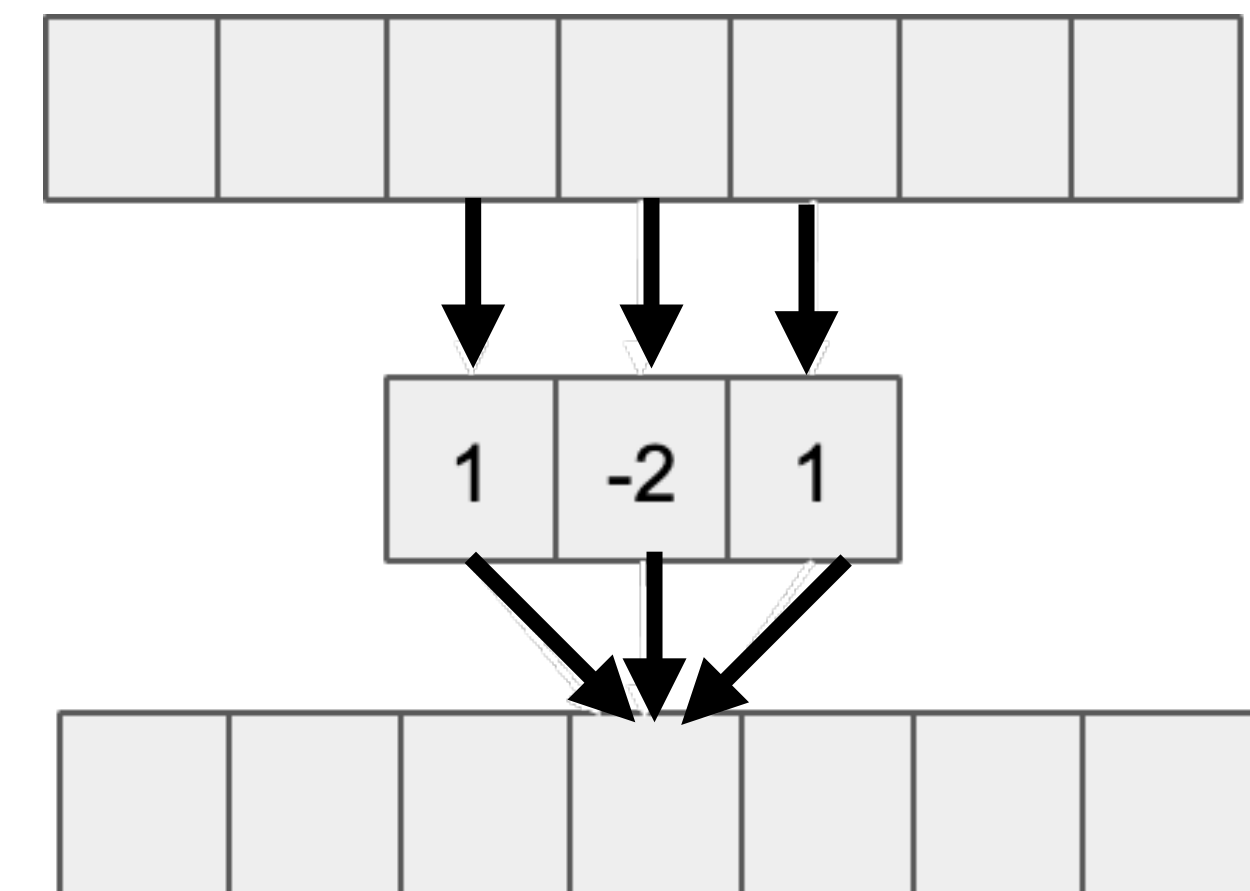


Accelerating Scientific Code: Raising Kernels to Convolutions

- Scientific codes (like state-of-the-art model, Oceananigans.jl) maintain hundreds of handwritten kernels, preventing them from using the advanced tensor capabilities of modern ML accelerators.
- Yet the core computations are often similar to ML (stencil kernel $\leftrightarrow$ convolution)

```
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
```



Accelerating Scientific Code

- New framework for raising and optimizing the structure within existing kernels to tensor IR!
 - 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 Oceananigans.jl 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

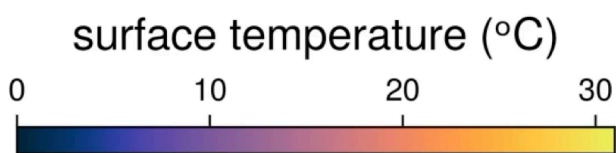
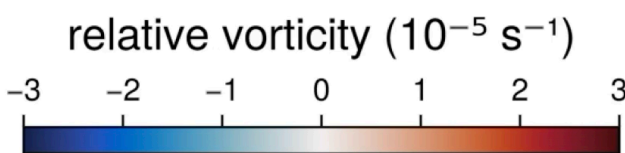
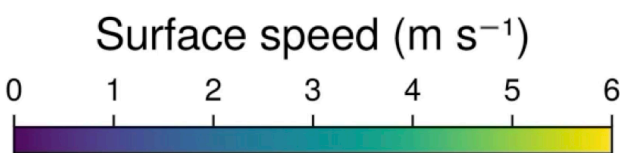
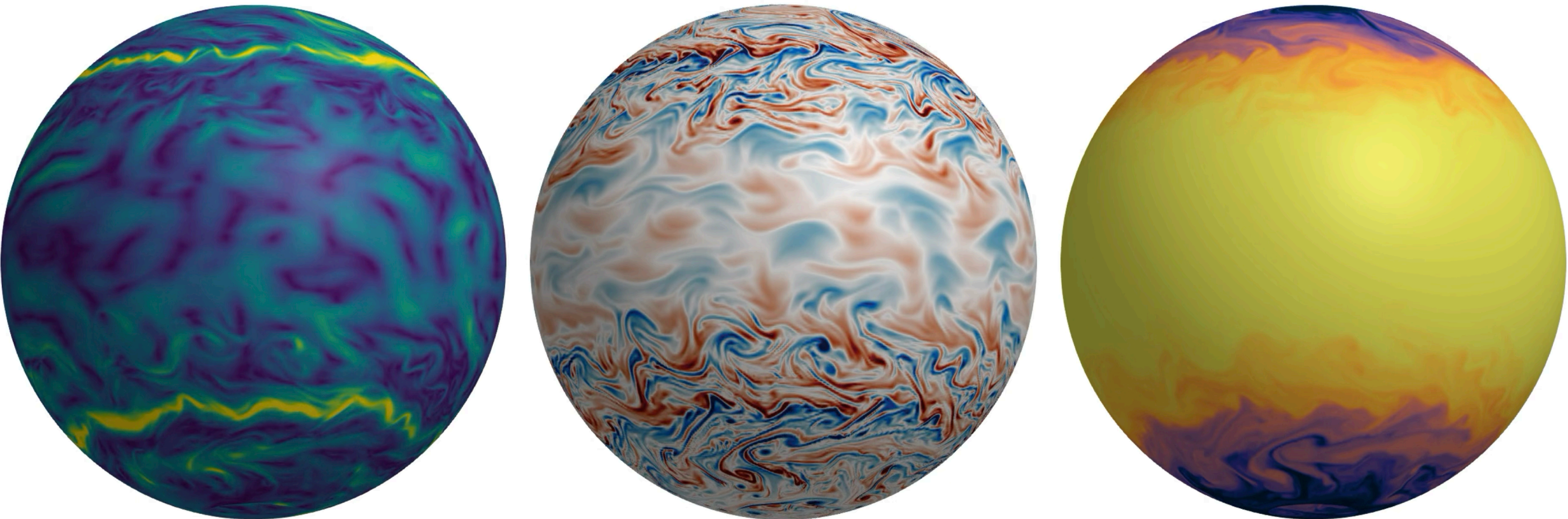
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

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


Accelerating Scientific Code

165.0 days



```
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]>
```

GPU Programming via LLVM

- Mainstream compilers do not have a high-level representation of parallelism, making optimization difficult or impossible
- This is accentuated for GPU programs where the kernel is kept in a separate module & synchronization is a barrier to optimization.

```
__global__ void normalize(int *out, int* in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```

Host Code

```
target triple = "x86_64-unknown-linux-gnu"  
  
define void @_Z6launchPiS_i(i32* %out,  
                           i32* %in,  
                           i32 %n) {  
    call i32 @pushCallConfiguration(...)  
    call i32 @cudaLaunch(@_device_stub, ...)  
    ret void  
}
```

Device Code

```
target triple = "nvptx"  
  
define void @_Z9normalize(i32* %out,  
                        i32* %in, i32 %n) {  
    %4 = call i32 @llvm.tid.x()  
    %5 = icmp slt i32 %4, %n  
    br i1 %5, label %6, label %13  
  
6:  
    %8 = getelementptr i32, i32* %in, i32 %4  
    %9 = load i32, i32* %8, align 4  
    %10 = call i32 @_Z3sumPii(i32* %in, i32 %n)  
    %11 = sdiv i32 %9, %10  
    %12 = getelementptr i32, i32* %out, i32 %4  
    store i32 %11, i32* %12, align 4  
    br label %13  
  
13:  
    ret void  
}
```


GPU Programming via MLIR

- Preserve Host & Device code through frontend (Plugin for C++, Julia Package)
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```



```
func @_Z6launch(%out: memref<?xi32>,  
               %in: memref<?xi32>, %n: i32) {  
    %c1 = constant 1 : index  
    %c0 = constant 0 : index  
  
    parallel (%tid) = (%c0) to (%n) step (%c1) {  
        %2 = load %in[%tid]  
        %sum = call @_Z3sumPii(%in, %n)  
        %4 = divsi %2, %sum : i32  
        store %4, %out[%tid]  
        yield  
    }  
    return  
}
```


GPU Programming via MLIR

- Preserve Host & Device code through frontend (Plugin for C++, Julia Package)
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```



```
func @_Z6launch(%out: memref<?xi32>,  
               %in: memref<?xi32>, %n: i32) {  
    %c1 = constant 1 : index  
    %c0 = constant 0 : index  
    %sum = call @_Z3sumPii(%in, %n)  
    parallel (%tid) = (%c0) to (%n) step (%c1) {  
        %2 = load %in[%tid]  
  
        %4 = divsi %2, %sum : i32  
        store %4, %out[%tid]  
        yield  
    }  
    return  
}
```

GPU Programming via MLIR

- Preserve Host & Device code through frontend (Plugin for C++, Julia Package)
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

Optimizations on primal => ***outsized impact for derivatives***

```
__global__ void normalize(int *out, int *in, int n) {  
    int tid = blockIdx.x;  
    if (tid < n)  
        out[tid] = in[tid] / sum(in, n);  
}  
  
void launch(int *out, int* in, int n) {  
    normalize<<<n>>>(d_out, d_in, n);  
}
```



```
func @_Z6launch(%out: memref<?xi32>,  
               %in: memref<?xi32>, %n: i32) {  
    %c1 = constant 1 : index  
    %c0 = constant 0 : index  
    %sum = call @_Z3sumPii(%in, %n)  
    parallel (%tid) = (%c0) to (%n) step (%c1) {  
        %2 = load %in[%tid]  
  
        %4 = divsi %2, %sum : i32  
        store %4, %out[%tid]  
        yield  
    }  
    return  
}
```

GPU Programming via MLIR

```
func @launch(%h_out : memref<?xf32>, %h_in : memref<?xf32>, %n : i64) {  
  parallel.for (%gx, %gy, %gz) = (0, 0, 0) to (grid.x, grid.y, grid.z) {  
    %shared_val = memref.alloca : memref<f32>  
    parallel.for (%tx, %ty, %tz) = (0, 0, 0) to (blk.x, blk.y, blk.z) {  
      if %tx == 0 {  
        store ..., %shared_val[] : memref<f32>  
      }  
      polygeist.barrier(%tx, %ty, %tz)  
      ...  
    }  
  }  
}
```


Synchronization via Memory

- Synchronization (`sync_threads`) ensures all threads within a block finish executing `codeA` before executing `codeB`
- The desired synchronization behavior can be reproduced by defining `sync_threads` to have the union of the memory semantics of the code before and after the sync.
- This prevents code motion of instructions which require the synchronization for correctness, but permits other code motion (e.g. index computation).

```
codeA(fib(idx));  
sync_threads;  
codeB(fib(idx));
```



```
off = fib(idx);  
codeA(off);  
sync_threads;  
codeB(off);
```

Synchronization via Memory

- High-level synchronization representation enables new optimizations, like sync elimination.
- A synchronize instruction is not needed if the set of read/writes before the sync don't conflict with the read/writes after the sync.

```
__global__ void bpnnp_layerforward(...) {
    __shared__ float node[HEIGHT];
    __shared__ float weights[HEIGHT][WIDTH];

    if ( tx == 0 )
        node[ty] = input[index_in] ;

    // Unnecessary Barrier #1
    // None of the read/writes below the sync
    // (weights, hidden)
    // intersect with the read/writes above the sync
    // (node, input)
    __syncthreads();

    // Unnecessary Store #1
    weights[ty][tx] = hidden[index];

    __syncthreads();

    // Unnecessary Load #1
    weights[ty][tx] = weights[ty][tx] * node[ty];
    ...
}
```


Synchronization via Memory

- Here on AMD GPUs, barrier synchronization is necessary because of the read/writes below the sync point, intersect with the read/writes above the sync point, and the read/writes after the sync point.
- 27% speedup on real code, 2.7x on PyTorch cross compilation!



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

William S. Moses
wmoses@mit.edu
MIT CSAIL
United States

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

Ivan R. Ivanov
ivanov@m.titech.ac.jp
Tokyo Tech
Japan

Johannes Doerfert
jdoerfert@lnl.gov
LLNL
United States

Jens Domke
jens.domke@riken.jp
RIKEN
Japan

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 58% 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.

CCS Concepts: • Software and its engineering → Compilers; • Theory of computation → Parallel computing models.

Keywords: Polygeist, MLIR, CUDA, Barrier Synchronization

ACM Reference Format:
William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *The 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23)*, February 25–March 1, 2023, Montreal, QC, Canada. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3572848.3577475>



This work is licensed under a Creative Commons Attribution International 4.0 License.
PPoPP '23, February 25–March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
<https://doi.org/10.1145/3572848.3577475>

Retargeting and Respecializing GPU Workloads for Performance Portability

Ivan R. Ivanov
Tokyo Institute of Technology
RIKEN R-CCS
Kobe, Japan
ivanov.i.aa@m.titech.ac.jp

Jens Domke
RIKEN R-CCS
Kobe, Japan
jens.domke@riken.jp

Toshio Endo
Tokyo Institute of Technology
Tokyo, Japan
endo@is.titech.ac.jp

William S. Moses
Google DeepMind
Paris, France
wsmoses@illinois.edu

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

Abstract—In order to come close to peak performance, accelerators like GPUs require significant architecture-specific tuning that understand the availability of shared memory, parallelism, tensor cores, etc. Unfortunately, the pursuit of higher performance and lower costs have led to a significant diversification of architecture designs, even from the same vendor. This creates the need for performance portability across different GPUs, especially important for programs in a particular programming model with a certain architecture in mind. Even when the program can be seamlessly executed on a different architecture, it may suffer a performance penalty due to it not being sized appropriately to the available hardware resources such as fast memory and registers, let alone not using newer advanced features of the architecture.

We propose a new approach to improving performance of (legacy) CUDA programs for modern machines by automatically adjusting the amount of work each parallel thread does, and the amount of memory and register resources it requires. By operating within the MLIR compiler infrastructure, we are able to also target AMD GPUs by performing automatic translation from CUDA and simultaneously adjust the program granularity to fit the size of target GPUs.

Combined with autotuning assisted by the platform-specific compiler, our approach demonstrates 27% geometric speedup on the Rodinia benchmark suite over baseline CUDA implementation as well as performance parity between similar NVIDIA and AMD GPUs executing the same CUDA program.

I. INTRODUCTION

Accelerators like GPUs remain the hardware target of choice for performance-critical software. Achieving high performance on these accelerators requires programmers to effectively leverage a peculiar programming model, most often exposed as C++ language extensions such as CUDA for NVIDIA GPUs and ROCm for AMD. While the community has developed alternative methods to portably program GPUs, including: a high-level block programming model in Triton [1], automatic mapping of C++ code onto GPUs [2], NumPy-style abstractions with varying degree of automated scheduling in JAX [3], TC [4], and TVM [5]; many of the performance-critical scientific

```
__void bpnnp_layerforward(...) {
    __float node[HEIGHT];
    __float weights[HEIGHT][WIDTH];

    == 0 )
    ty] = input[index_in] ;

    // Unnecessary Barrier #1
    // of the read/writes below the sync
    // points, hidden)
    // intersect with the read/writes above the sync
    // point, input)
    threads();

    // Unnecessary Store #1
    [ty][tx] = hidden[index];

    __syncthreads();

    // Unnecessary Load #1
    weights[ty][tx] = weights[ty][tx] * node[ty];

    ...
}
```


Synchronization via Memory

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. historically OpenMP, now TPUs)
- Some backends do not have block synchronization
- Lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

```
parallel_for %i = 0 to N {  
  codeA(%i);  
  sync_threads;  
  codeB(%i);  
}
```



```
parallel_for %i = 0 to N {  
  codeA(%i);  
}  
parallel_for %i = 0 to N {  
  codeB(%i);  
}
```

Synchronization via Memory

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. historically OpenMP, now TPUs)
- Some backends do not have block synchronization
- Lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

```
parallel_for %i = 0 to N {  
  for %j = ... {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}
```



```
for %j = ... {  
  parallel_for %i = 0 to N {  
    codeB1(%i, %j);  
    sync_threads;  
    codeB2(%i, %j);  
  }  
}
```

Reactant + Millimeter Wave Association

- Generate mock galaxy catalog, add a Spectral Energy Distribution to each galaxy, Calculate fluxes in each filter of interest
- Reactant-compiled model beats prior workloads by an order of magnitude thanks to auto-parallelization and linear algebra transformations
- Integration of HMC in progress

