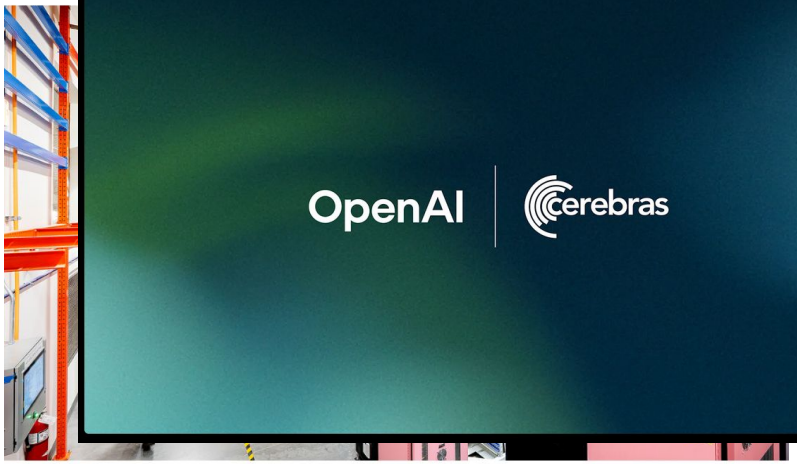
Accelerating scientific code on AI hardware with Reactant.jl

Mosè Giordano (UCL) & Jules Merckx (Ghent University)

2026-02-01 @ FOSDEM '26

AI hardware dominates modern computing world

AWS activates Project Rainier: One of the world's largest AI compute clusters comes online



Written by Amazon Staff

Last updated: October 29, 2025

6 min read

January 14, 2026 Global Affairs

OpenAI partners with Cerebras

OpenAI is partnering with Cerebras to add 750MW of ultra low-latency AI compute to our platform.

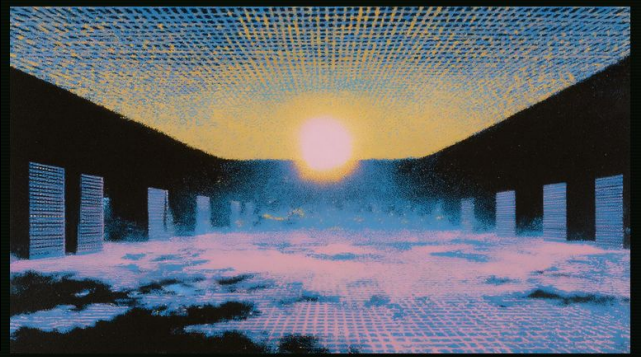
NNOUNCEME

AMD

xAI Colossus Hits 2 GW: 555,000 GPUs, $18B, Largest AI Site

Musk's xAI purchases third Memphis building for 2 GW total capacity. 555,000 NVIDIA GPUs at $18B makes Colossus the world's largest AI training facility.

Blake Crosley · Jan 03, 2026 · 4 min read · Disclaimer



January 1, 2026

tts of AMD

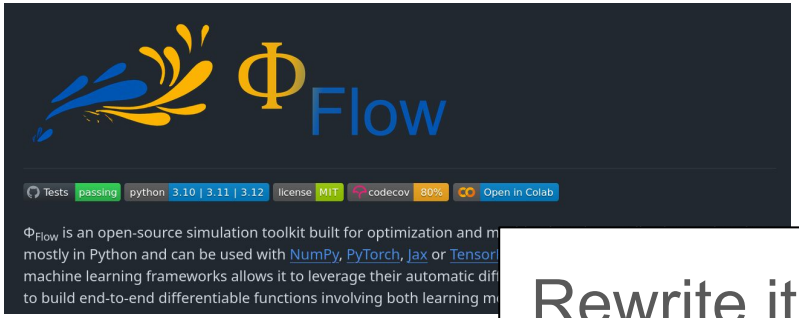
AI on alert and

lvidia's stock and prompt

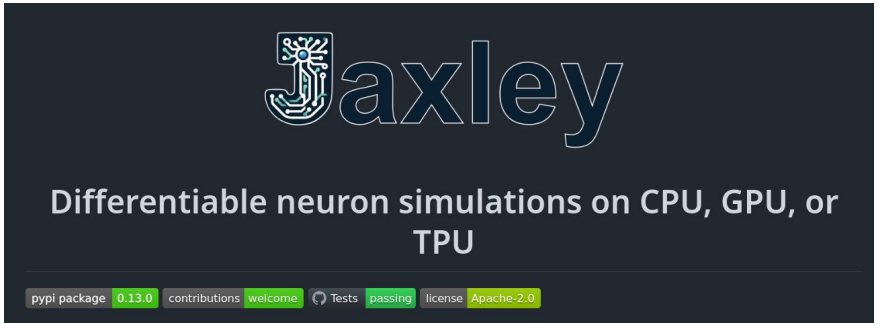
v Gemini 3 AI models, which have
ontage/Getty Images

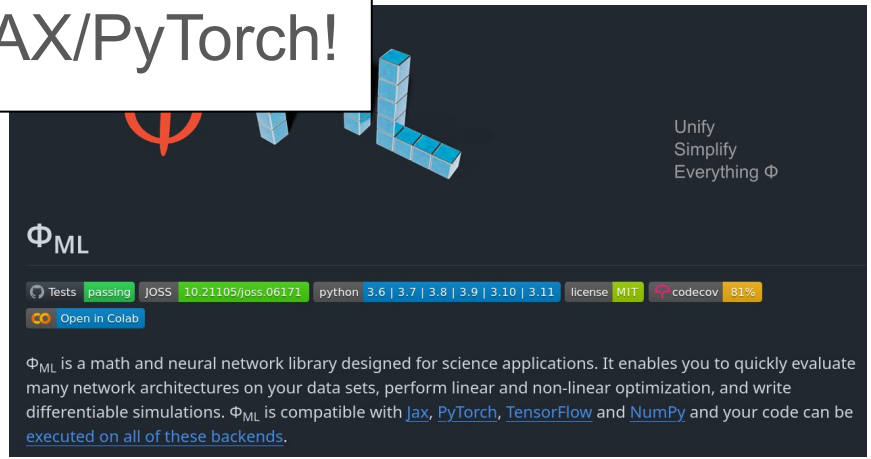
58

How do we write scientific code for ML accelerators?



Rewrite it in JAX/PyTorch!

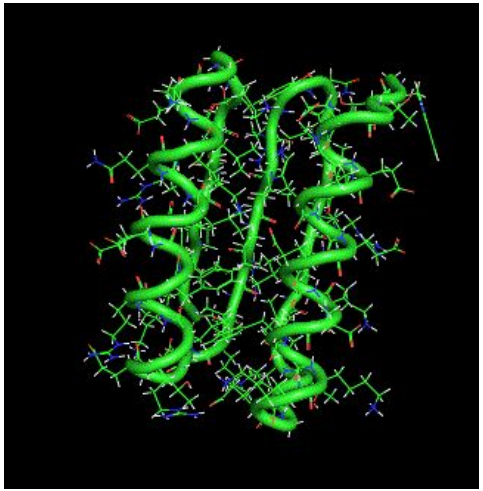




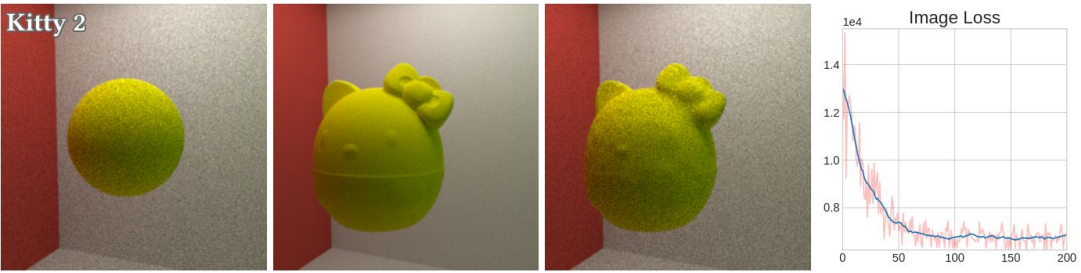
Differentiation: connecting science and AI

Derivatives are key to science + ML

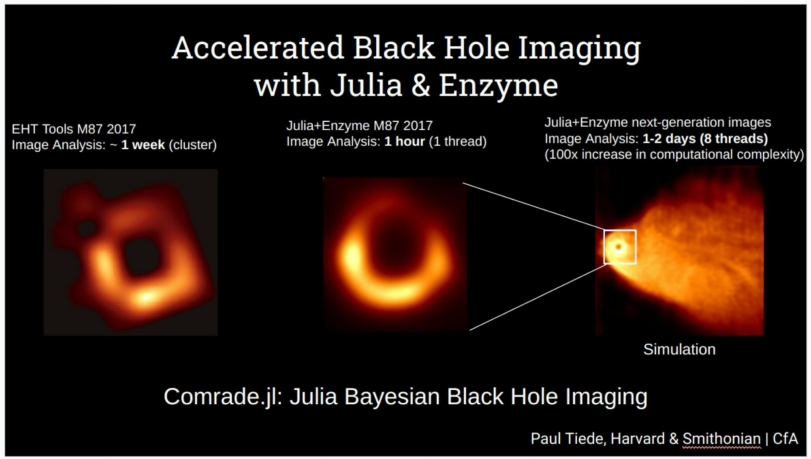
- *Scientific Computing*: UQ, Differential Equation, Error Analysis
- *Machine Learning*: Back-Propagation, Bayesian Inference



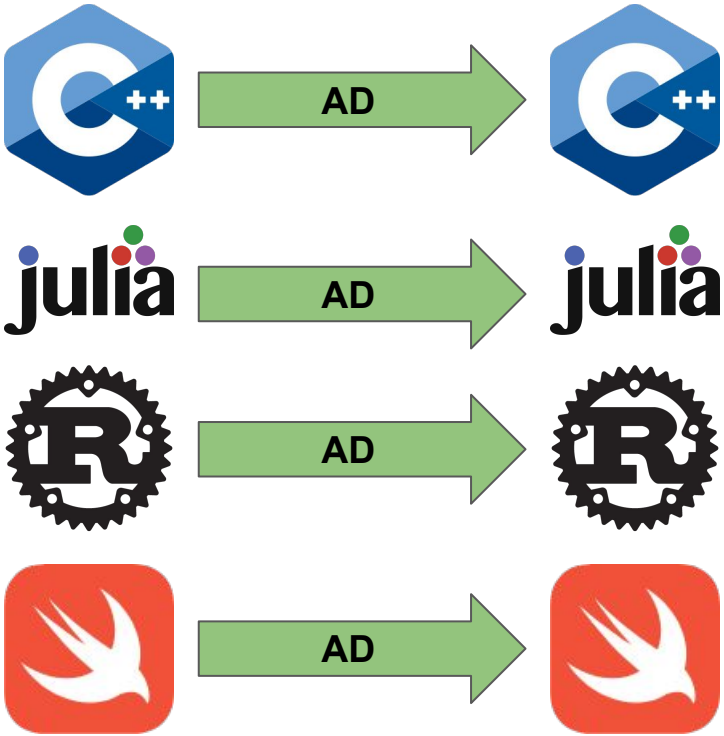
Foldit1 (PDB ID 6MRR) simulated with Molly.jl in the a99SB-ILDN force field with explicit solvent, (Greener)

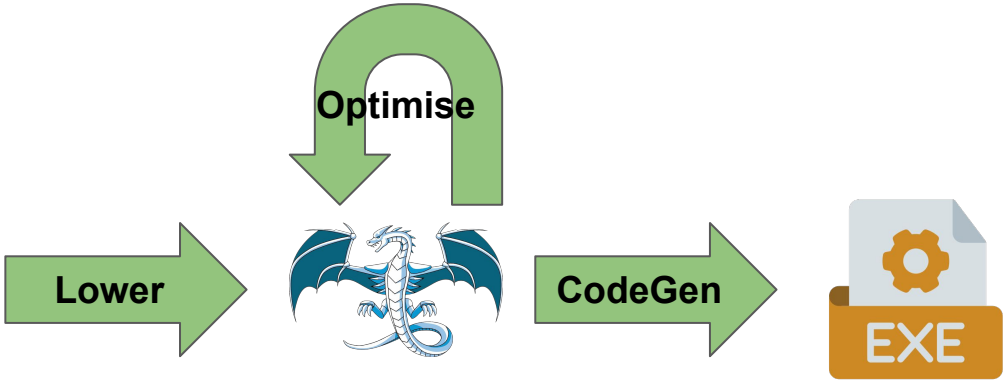


Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering (Zihan Yu et al)

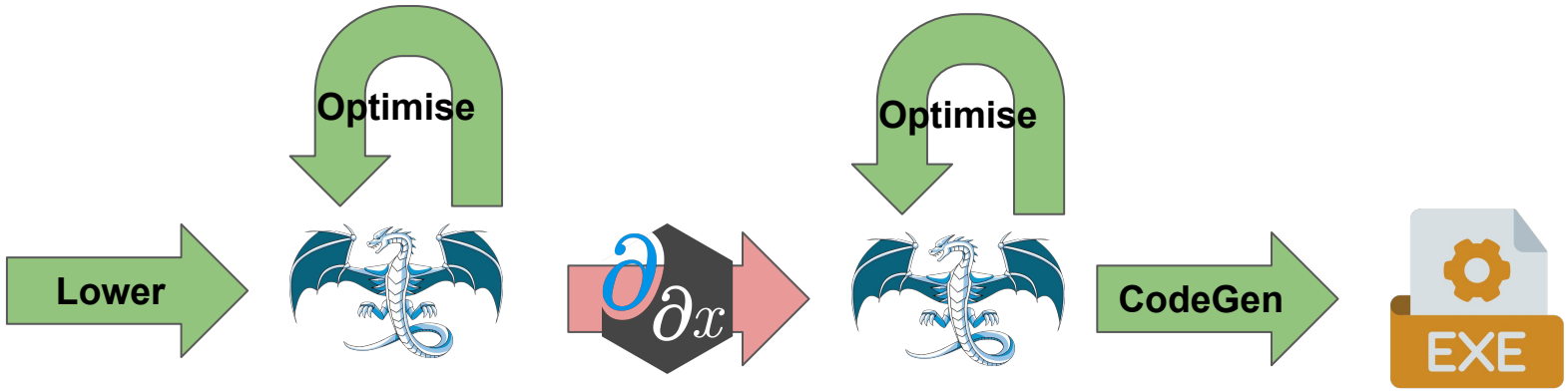


Most automatic differentiation pipelines





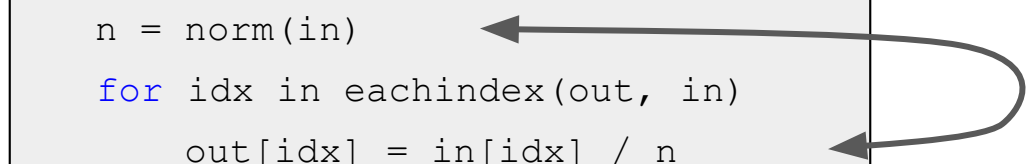
Autodiff: the Enzyme approach



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

Case study: vector normalization

```
# Compute `normalize!` in O(n)
function normalize!(out, in)
    n = norm(in)
    for idx in eachindex(out, in)
        out[idx] = in[idx] / n
    end
end
```



Optimisation & automatic differentiation

$O(n^2)$

```
for idx in eachindex(out, in)
   out[idx] = in[idx] / norm(in)
end
```

AD

$O(n^2)$

```
for idx in eachindex(d_out)
   d_n = d_out[idx]...
   ∇normalize!(d_n, d_in)
end
```

Optimise

$O(n^2)$

```
for idx in eachindex(d_out)
   d_n = d_out[idx]...
   ∇normalize!(d_n, d_in)
end
```

$O(n^2)$

```
for idx in eachindex(out, in)
   out[idx] = in[idx] / norm(in)
end
```

Optimise

$O(n)$

```
n = norm(in)
for idx in eachindex(out)
   out[idx] = in[idx] / n
end
```

$\frac{\partial}{\partial x}$

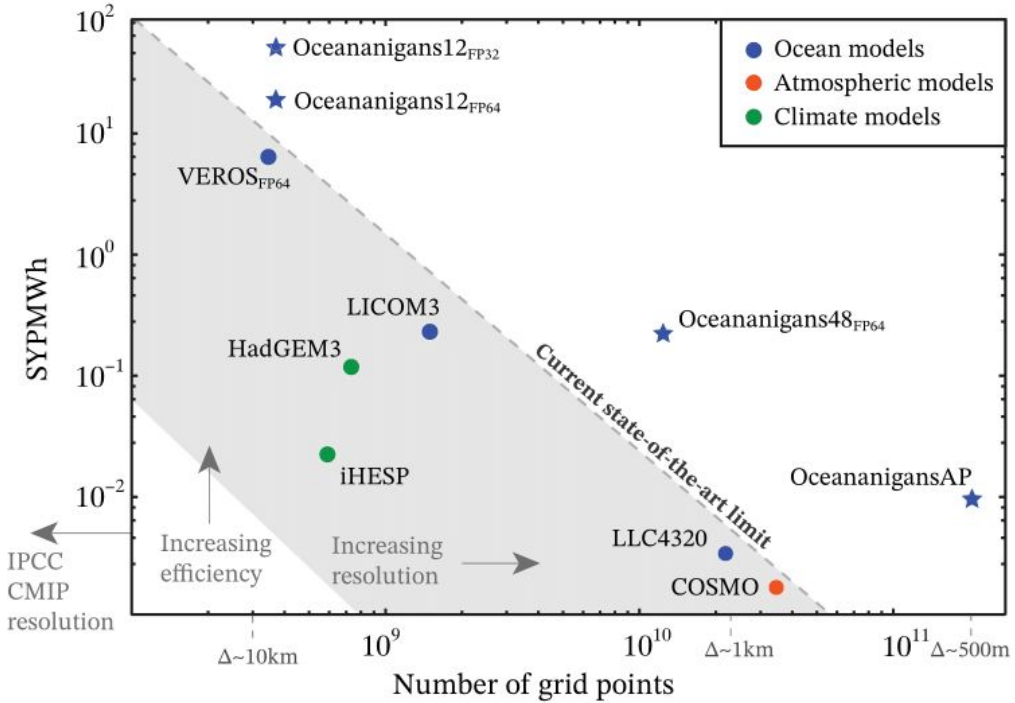
$O(n)$

```
d_n = 0.0
for idx in eachindex(d_out)
   d_n += d_out[idx]...
end
∇normalize!(d_n, d_in)
```

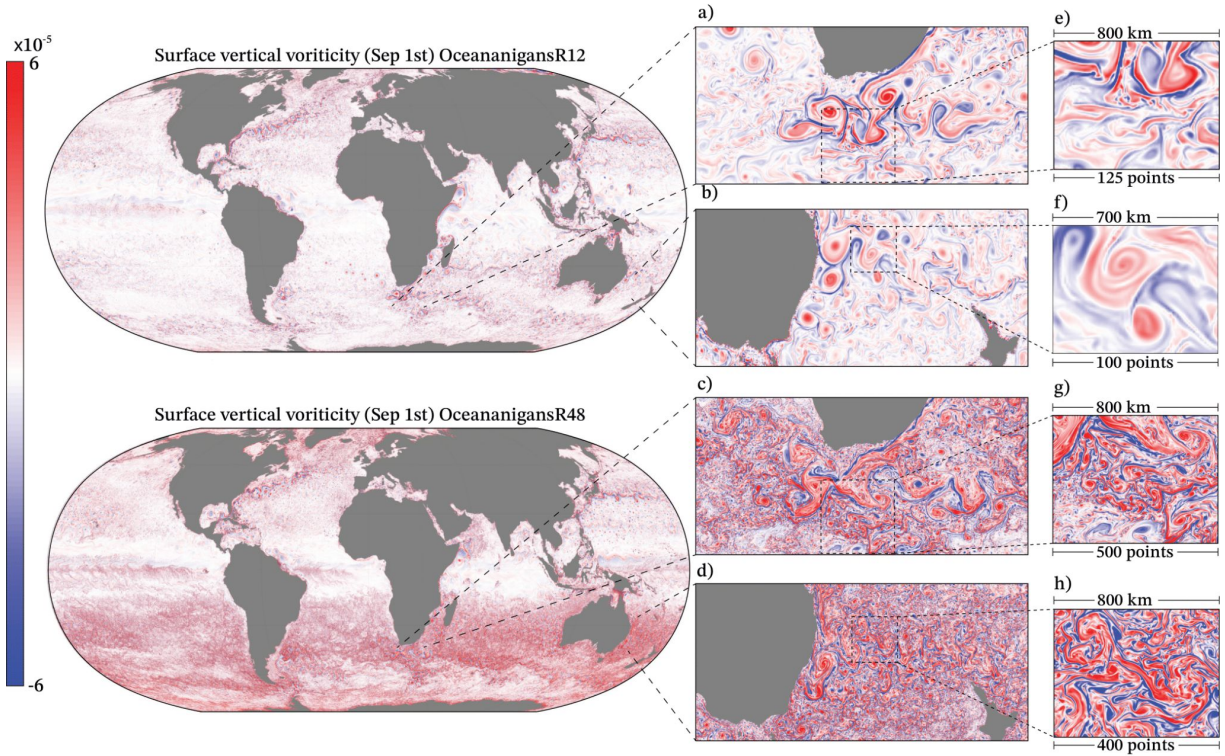
Science problem: Oceananigans

Oceananigans is a fast, friendly, flexible software package for **finite volume simulations** of the **nonhydrostatic and hydrostatic Boussinesq equations** written in Julia.

Already used on up to 768 GPUs, demonstrating high memory and energy efficiency: https://arxiv.org/abs/2309.06662



Science problem: Oceananigans



GPU programming via LLVM

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

```
__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);
}
```

Device code

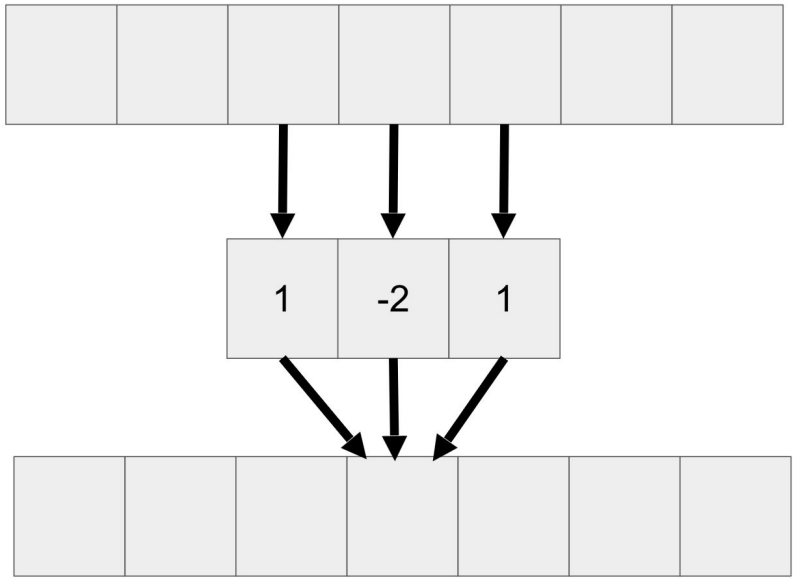
Host code

```
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
}
```

```
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
}
```

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
```



Over 270 such kernels in Oceananigans

GPU programming via MLIR

- Preserve **host & device code** through frontend (Clang plugin for C++, JIT package for Julia, etc)
- Enables **optimisation between caller and kernel**
- Enable parallelism-specific optimisation
- Optimisations on primal =>
  outsized impact for derivatives

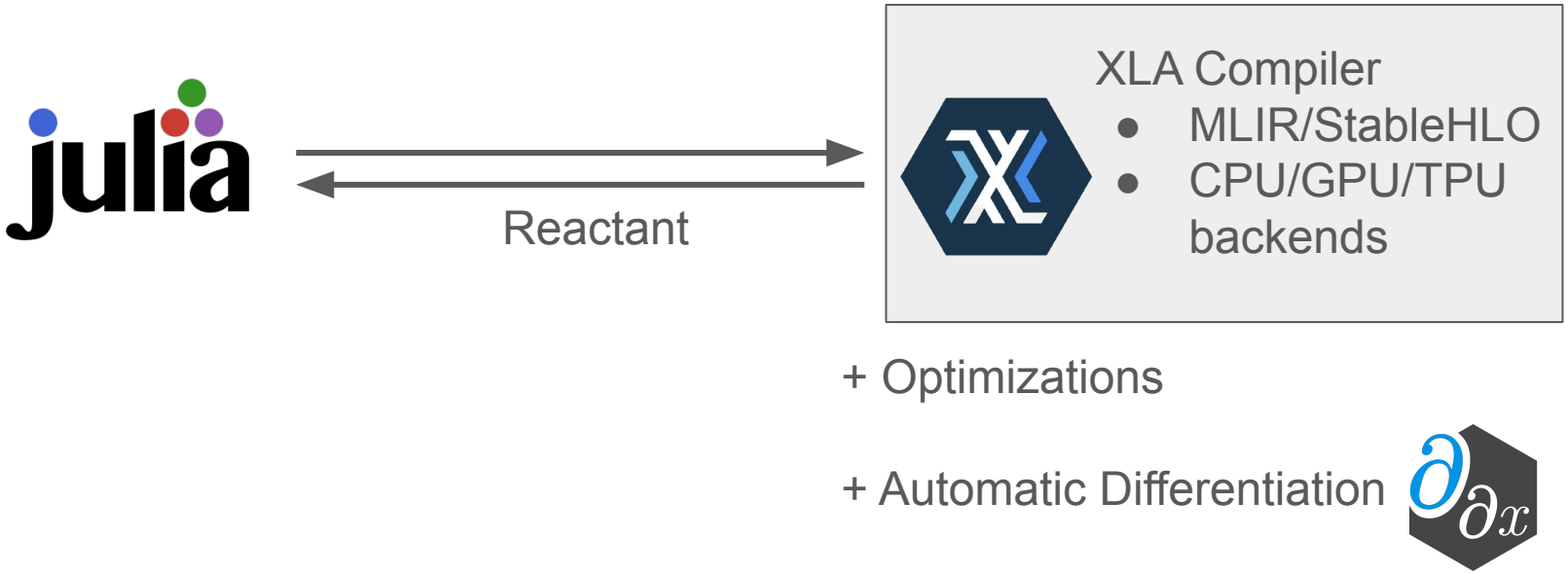
```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}
void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
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
}
```

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

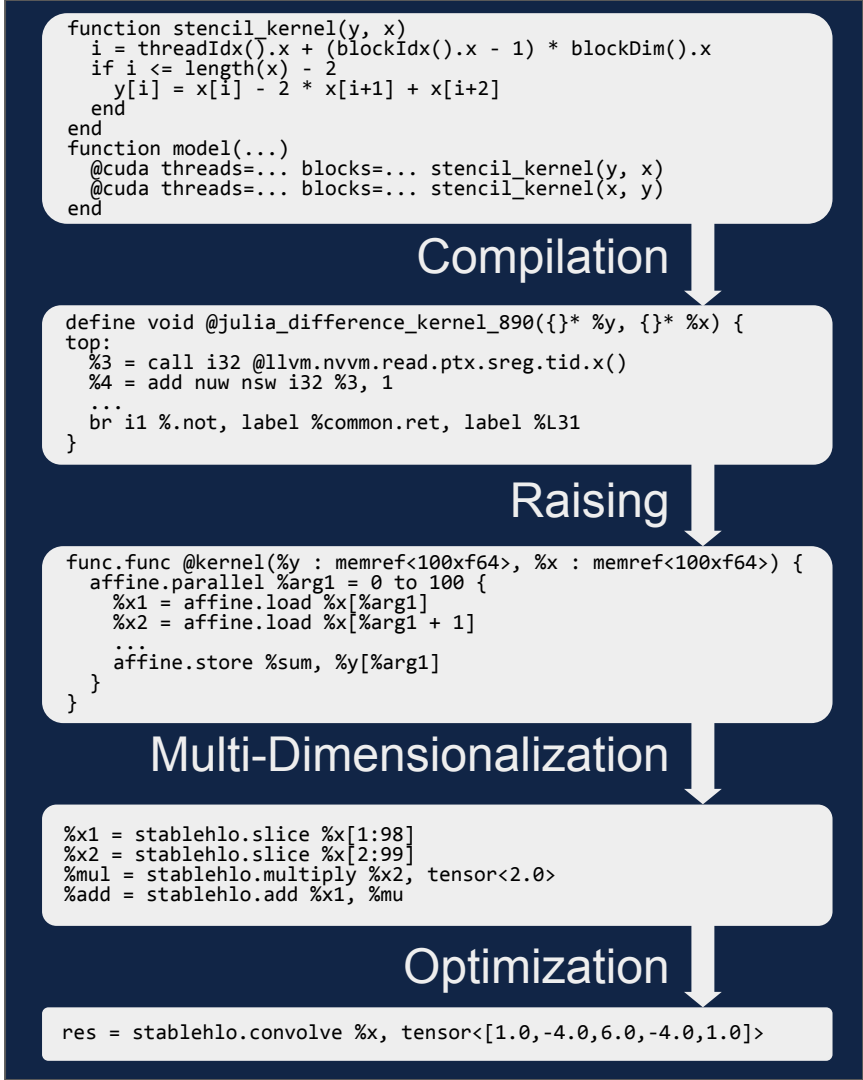
Reactant.jl

“Optimize Julia Functions With MLIR and XLA for High-Performance Execution on CPU, GPU, TPU and more.”



CUDA to accelerator IR (StableHLO)

- New framework for raising and optimising 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. optimise
- Compiled single-node CUDA version of code to execute on thousands of distributed TPUs and GPUs

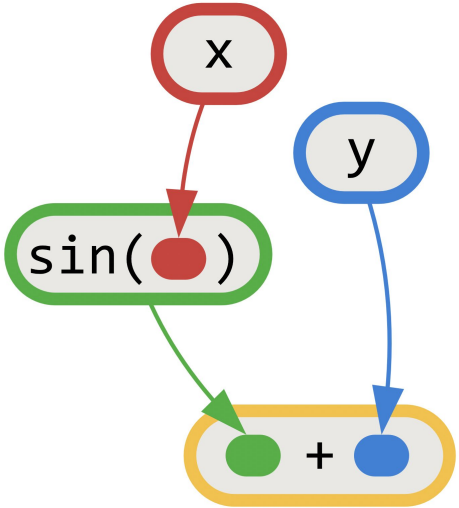


Julia code is traced to generate stablehlo

```julia
function f(x, y)
    sin.(x) .+ y
end
```

```julia
result = @jit f(
  to_rarray(rand(10, 20)),
  to_rarray(rand(1 , 20))
)
```

Create Reactant arrays




```
func.func @f(%x: tensor<20x10xf64>, %y: tensor<20x1xf64>) -> tensor<20x10xf64> {
  %0 = stablehlo.sine %x : tensor<20x10xf64>
  %1 = stablehlo.broadcast_in_dim %y, dims = [0, 1] : (tensor<20x1xf64>) -> tensor<20x10xf64>
  %2 = stablehlo.add %0, %1 : tensor<20x10xf64>
  return %2 : tensor<20x10xf64>
}
```

Reactant supports arbitrary Julia structs as arguments

```
distance(a::Point, b::Point) = sqrt((a.x - b.x)^2 + (a.y - b.y)^2)
```

```
struct Point{T}
  x::T
  y::T
end
```

```
func.func @distance(
    %a_x: tensor<f64>, %a_y: tensor<f64>,
    %b_x: tensor<f64>, %b_y: tensor<f64>
) -> tensor<f64> {
  %0 = stablehlo.subtract %a_x, %b_x : tensor<f64>
  %1 = stablehlo.multiply %0, %0 : tensor<f64>
  %2 = stablehlo.subtract %a_y, %b_y : tensor<f64>
  %3 = stablehlo.multiply %2, %2 : tensor<f64>
  %4 = stablehlo.add %1, %3 : tensor<f64>
  %5 = stablehlo.sqrt %4 : tensor<f64>
  return %5 : tensor<f64>
}
```

Reactant can trace control flow*

```
function nnorm(x, n)
  @trace for i in 1:n
    x = x * i ./ sum(x)
  end
  return x
end
```

```
func.func @main(%x: tensor<10xf64>, %n: tensor<i64>) -> tensor<10xf64> {
  %cst = stablehlo.constant dense<0.0> : tensor<f64>
  %i_0 = stablehlo.constant dense<0> : tensor<i64>
  %step = stablehlo.constant dense<1> : tensor<i64>
  %0:2 = stablehlo.while(%i = %i_0, %x_inner = %x) : tensor<i64>, tensor<10xf64> cond {
    %1 = stablehlo.compare  LT, %i, %n : (tensor<i64>, tensor<i64>) -> tensor<i1>
    stablehlo.return %1 : tensor<i1>
  } do {
    %i_new = stablehlo.add %step, %i : tensor<i64>
    ...
  }
  return %0#1 : tensor<10xf64>
}
```

Traced Julia code can mutate arrays

```
function set_zero!(x::AbstractArray{T}) where T
  x[1, 1] = zero(T)
  return nothing
end
```

stablehlo IR cannot express mutation

```
func.func @main(%x: tensor<10x10xf64>) -> tensor<10x10xf64> {
  %zero = stablehlo.constant dense<0.000000e+00> : tensor<1x1xf64>
  %i = stablehlo.constant dense<0> : tensor<i32>
  %x_new = stablehlo.dynamic_update_slice %x, %zero, %i, %i : (...) -> tensor<10x10xf64>
  return %x_new : tensor<10x10xf64>
}
```

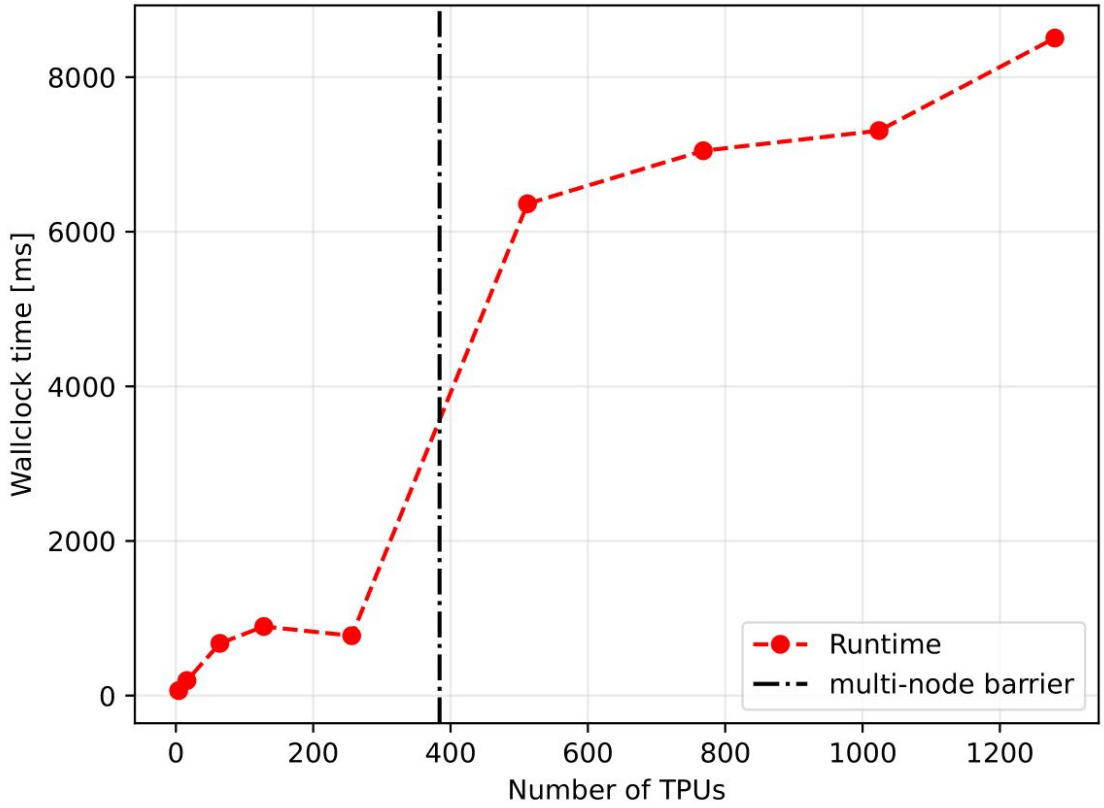
Updated value is returned

…and more

- Generated stablehlo IR is heavily optimized
- Automatic distribution via sharding or MPI (talk to us to hear more)
- Automatic differentiation using Enzyme
- Julia frontend, C++ frontend, and raising pipeline
- …

More about Reactant.jl: Reactant: Optimize Julia functions with MLIR & XLA @ JuliaCon ‘25 by William Moses

Large scale run on TPUs



Conclusions

- Reactant.jl is an **optimising compiler framework for Julia**, built on top of MLIR
- It **preserves high-level structures** from Julia code into the compiler
- Effectively **optimise generic scientific programs**, doesn't require a DSL
- Effectively **retarget programs**, allowing them to run on distributed clusters of your favorite accelerator, including CPU/GPU (Nvidia, AMD)/TPU/TensTorrent, more accelerators to come in the future
- Can **leverage XLA** (state of the art runtime, used by TensorFlow, JaX, & PyTorch)
- Compatible with **mutation**, **control flow** (with caveats), **autodiff** (Enzyme)
- Backend (Enzyme-JAX) can be used also in **C++ and Python**
    - Has demonstrated ~50% performance improvement in Google Gemma model on 32 accelerators

All code is **open source** (https://github.com/EnzymeAD/Reactant.jl), and you can play yourself with it on Google Colab (both TPUs and GPUs)