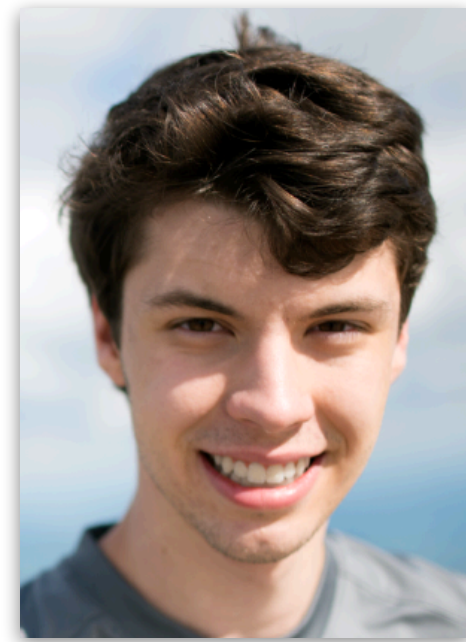




# Enzyme: High-Performance, Cross-Language, and Parallel Automatic Differentiation



William S. Moses

[wsmoses@illinois.edu](mailto:wsmoses@illinois.edu)

LANL Seminar

May 27, 2026



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN



William S. Moses<sup>†§</sup>, Mosè Giordano<sup>★</sup>, Avik Pal<sup>‡</sup>, Gregory Wagner<sup>‡</sup>, Ivan R Ivanov, Paul Berg<sup>∇</sup>,  
Johannes Blaschke, Jules Merckx<sup>△</sup>, Arpit Jaiswal<sup>◆</sup>, Patrick Heimbach<sup>#</sup>, Son Vu, Sergio  
Sanchez-Ramirez<sup>◇</sup>, Simone Silvestri, Nora Loose<sup>♣</sup>, Ivan Ho, Vimarsh Sathia<sup>†</sup>, Jan Hueckelheim<sup>♣</sup>,  
Johannes De Fine Licht, Kevin Gleason<sup>§</sup>, Ludovic Rass, Gabriel Baraldi, Dhruv Apte<sup>#</sup>, Lorenzo  
Chelini<sup>◆</sup>, Jacques Pienaar<sup>§</sup>, Gaetan Lounes, Valentin Churavy, Sri Hari Krishna Narayanan<sup>♣</sup>, Navid  
Constantinou, William R. Magro<sup>§</sup>, Michel Schanen<sup>♣</sup>, Alexis Montoison<sup>♣</sup>, Alan Edelman<sup>‡</sup>, Samarth  
Narang, Tobias Grosser, Keno Fischer<sup>‡</sup>, Robert Hundt<sup>§</sup>, Albert Cohen<sup>§</sup>, Oleksandr Zinenko<sup>§ \*</sup>  
UIUC<sup>†</sup>, Google<sup>§</sup>, UCL<sup>★</sup>, MIT<sup>‡</sup>, NVIDIA<sup>◆</sup>, UT Austin<sup>#</sup>, [C]Worthy<sup>♣</sup>, BSC<sup>◇</sup>, Argonne National Laboratory<sup>♣</sup>,  
LBNL<sup>♡</sup>, Cambridge<sup>‡</sup>, JuliaHub<sup>‡</sup>, University of Mainz<sup>#</sup>, BFH<sup>∇</sup>, Ghent University<sup>△</sup>

# Outline

---

- Compiler-Based Differentiation (EnzymeLLVM)



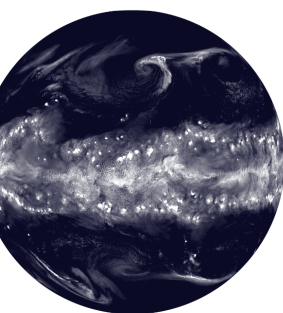
- Parallel Differentiation (CPUs, GPUs, MPI, etc)?



- Differentiate & Optimize User-Defined Operations (EnzymeMLIR)



- Run and Differentiate Existing Code on Thousands of Accelerators (Reactant)

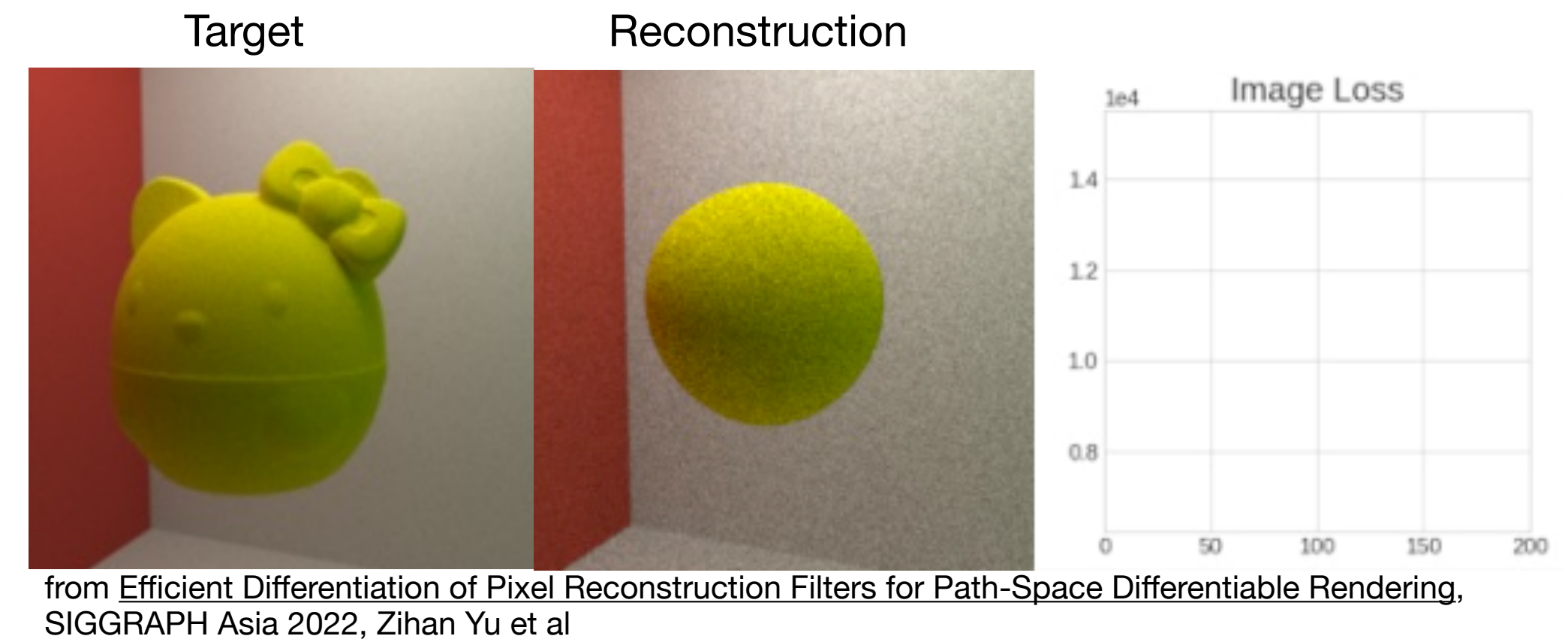




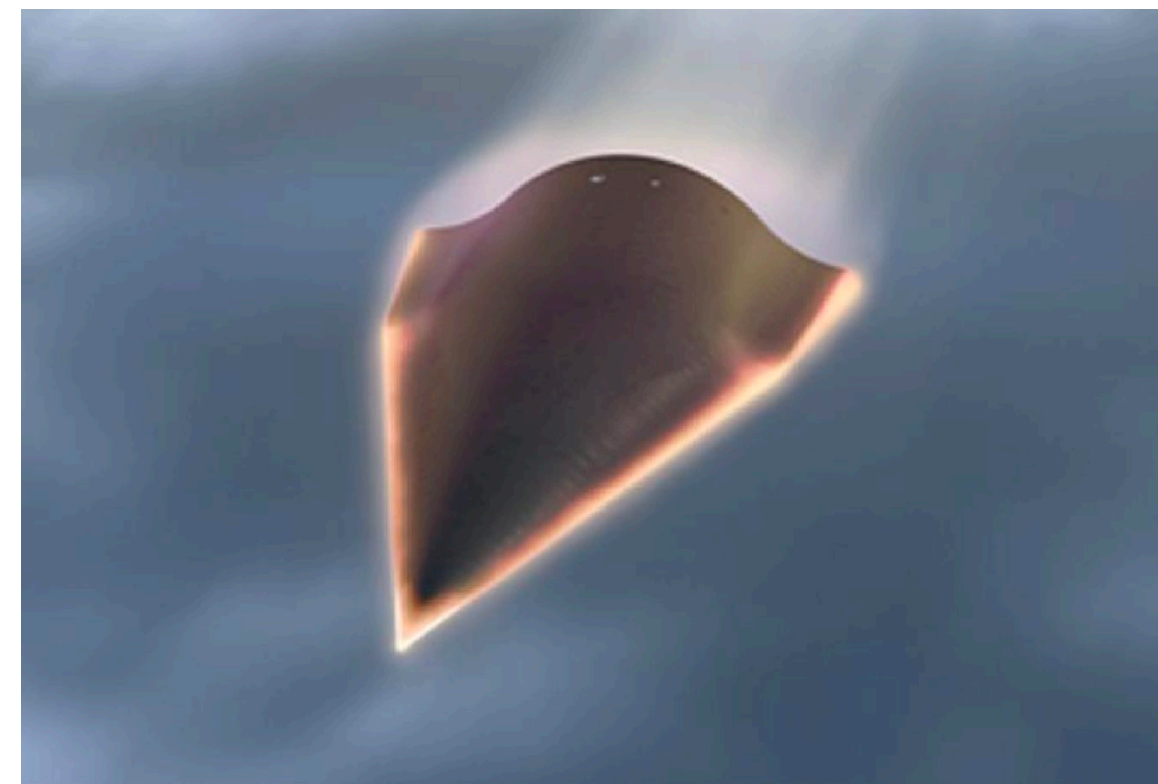
# Differentiation: Connecting Science and AI

Derivatives are key to science + ML

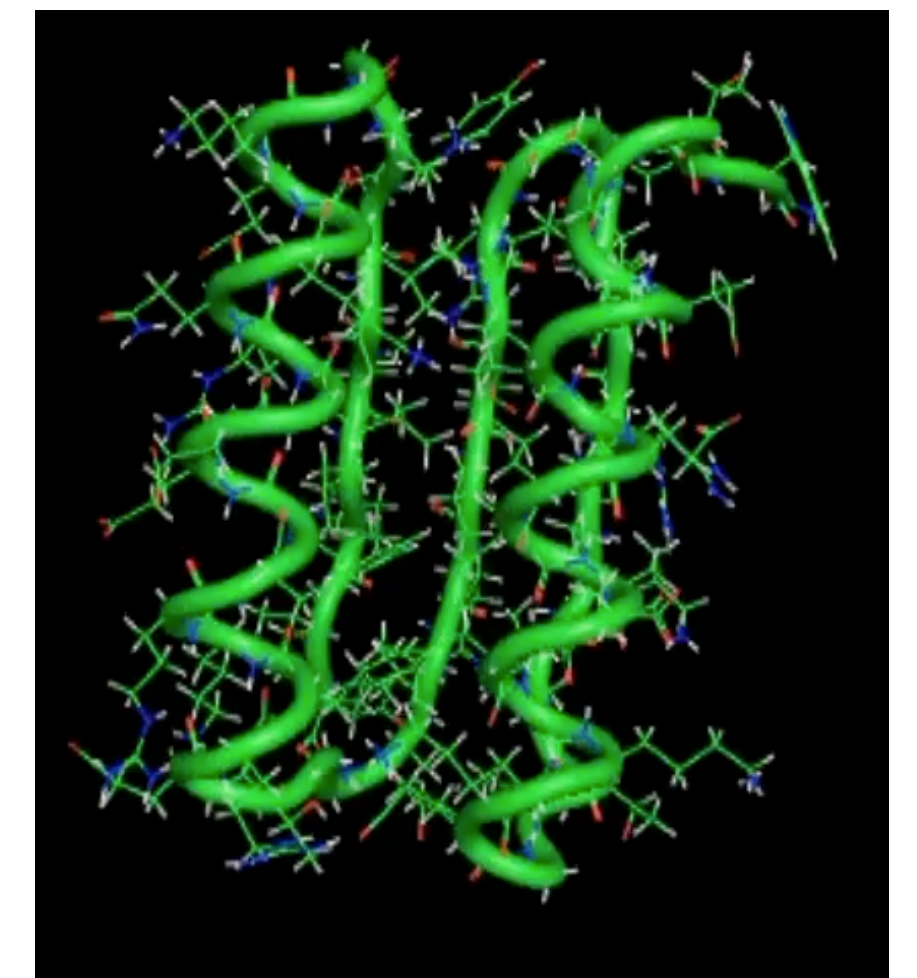
- Scientific Computing: UQ, Differential Equation, Error Analysis
- Machine Learning: Back-Propagation, Bayesian Inference



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



from Center for the Exascale Simulation of Materials in Extreme Environments

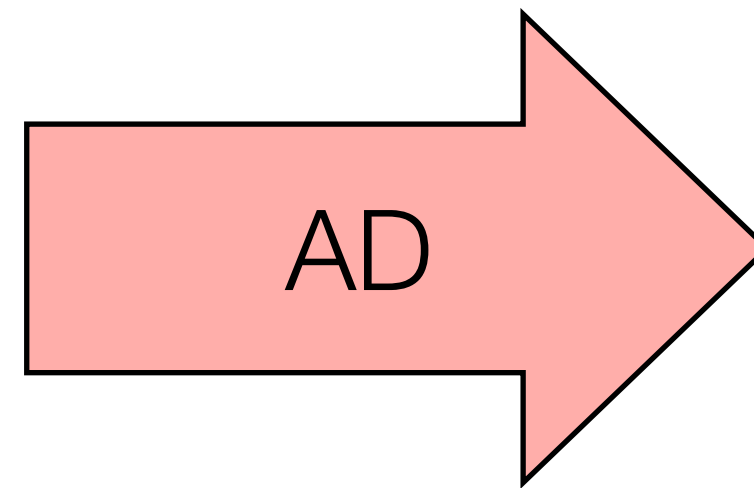


from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

# Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs

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



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

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

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

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

# Differentiation is Expensive

---

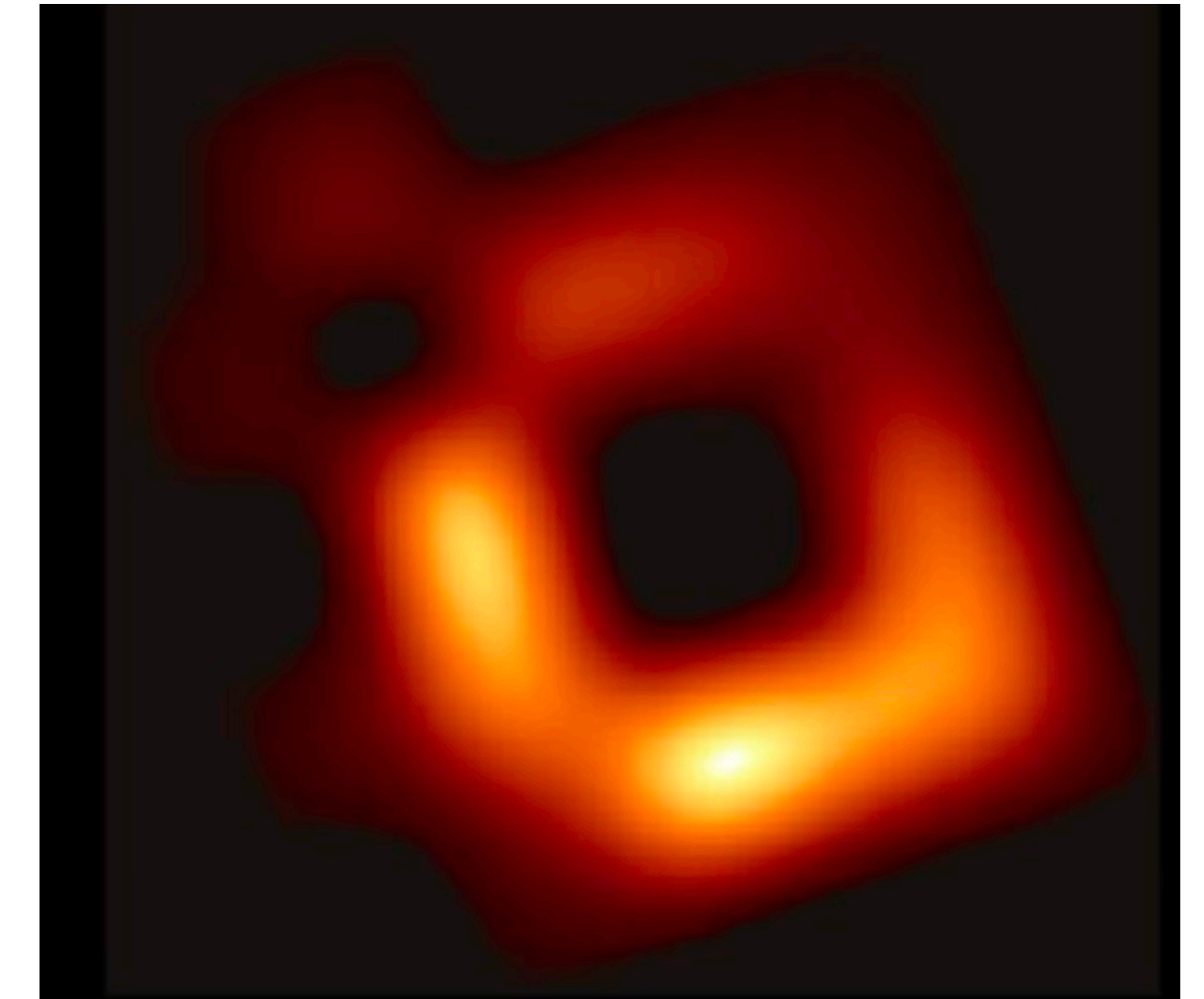
Derivatives are the most costly and difficult to use algorithms

# Differentiation is Expensive

---

Derivatives are the most costly and difficult to use algorithms

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

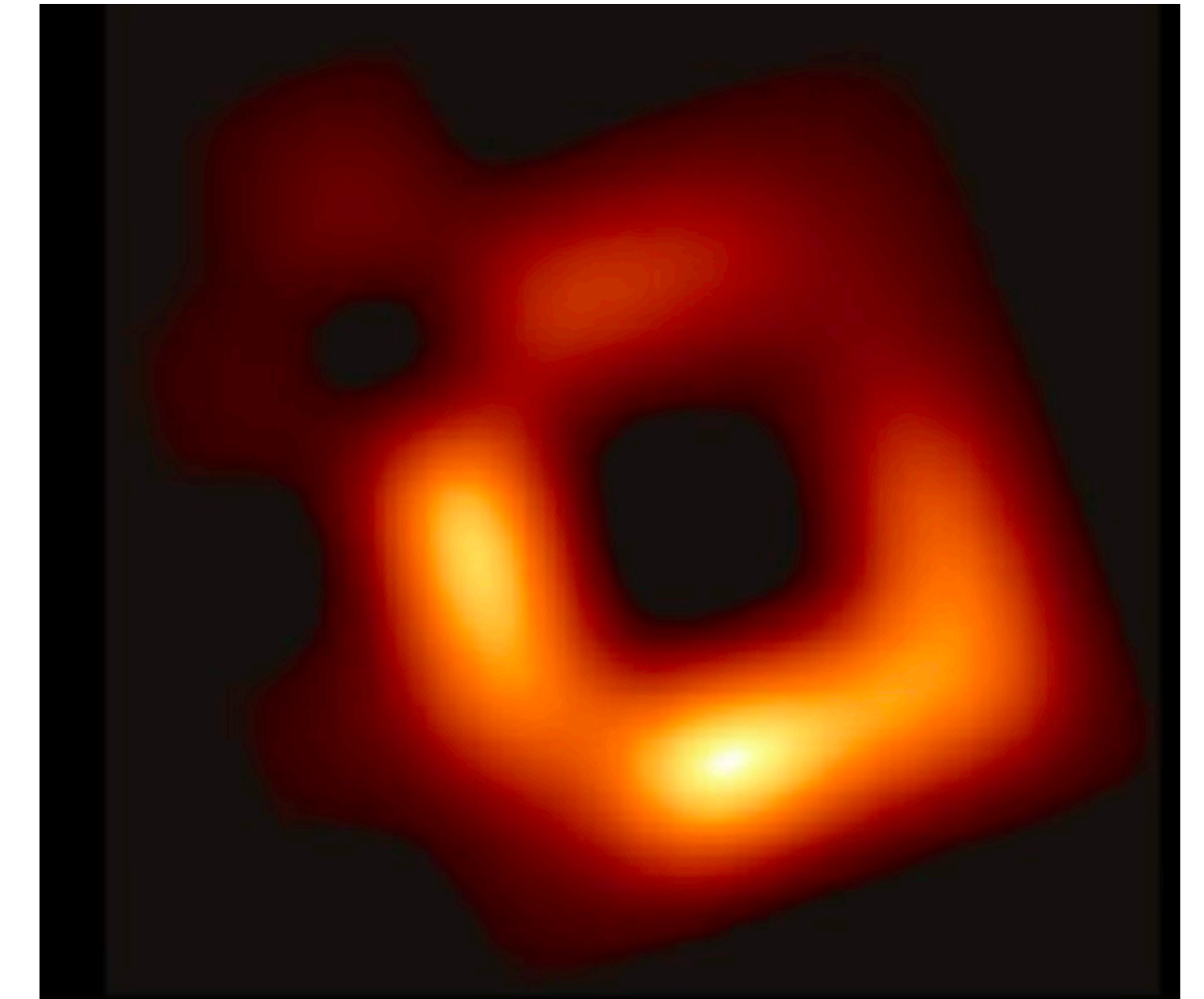


# Differentiation is Expensive

---

Derivatives are the most costly and difficult to use algorithms

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



With Enzyme differentiation:  
1 hour on 1 thread

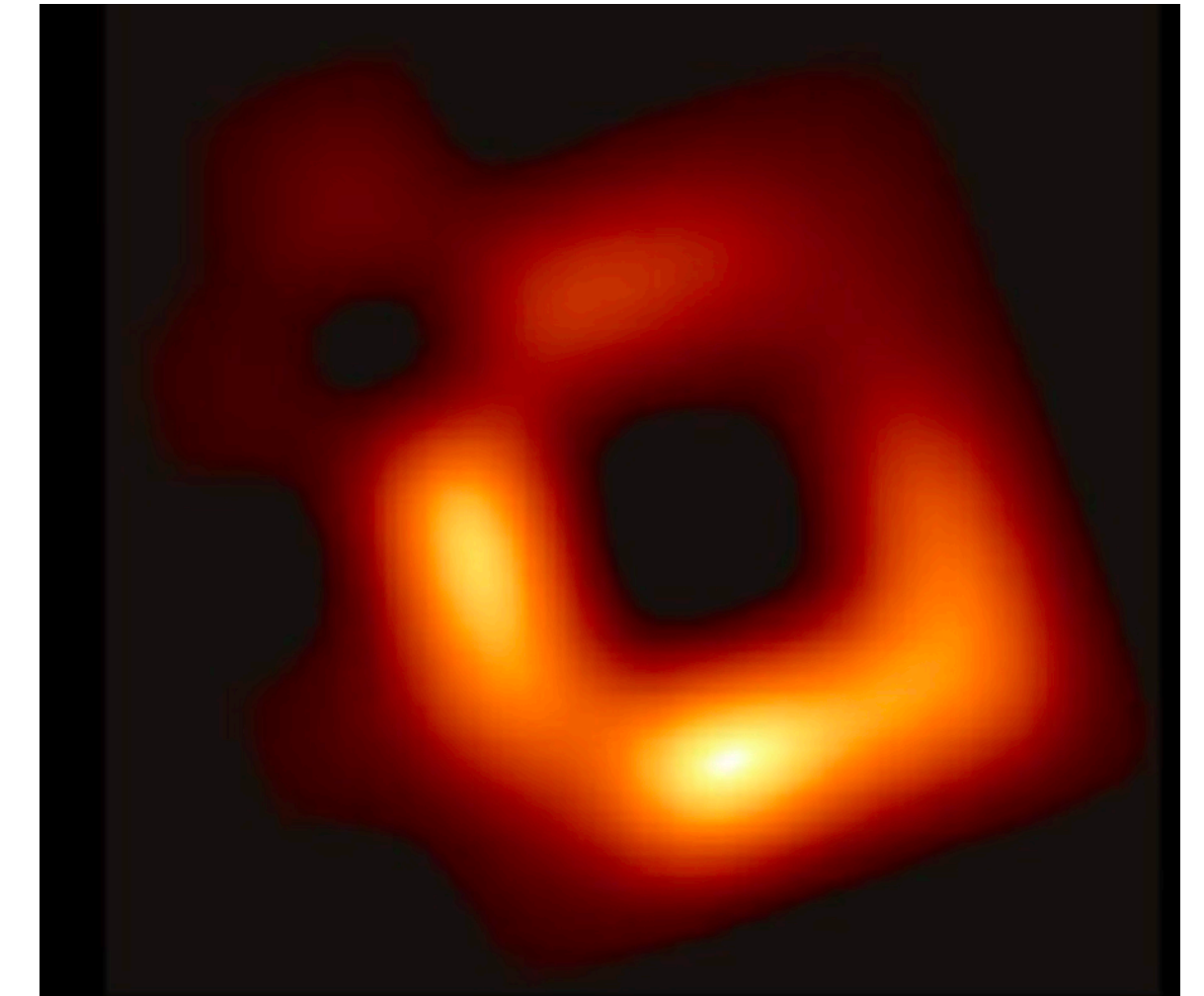


# Differentiation is Expensive

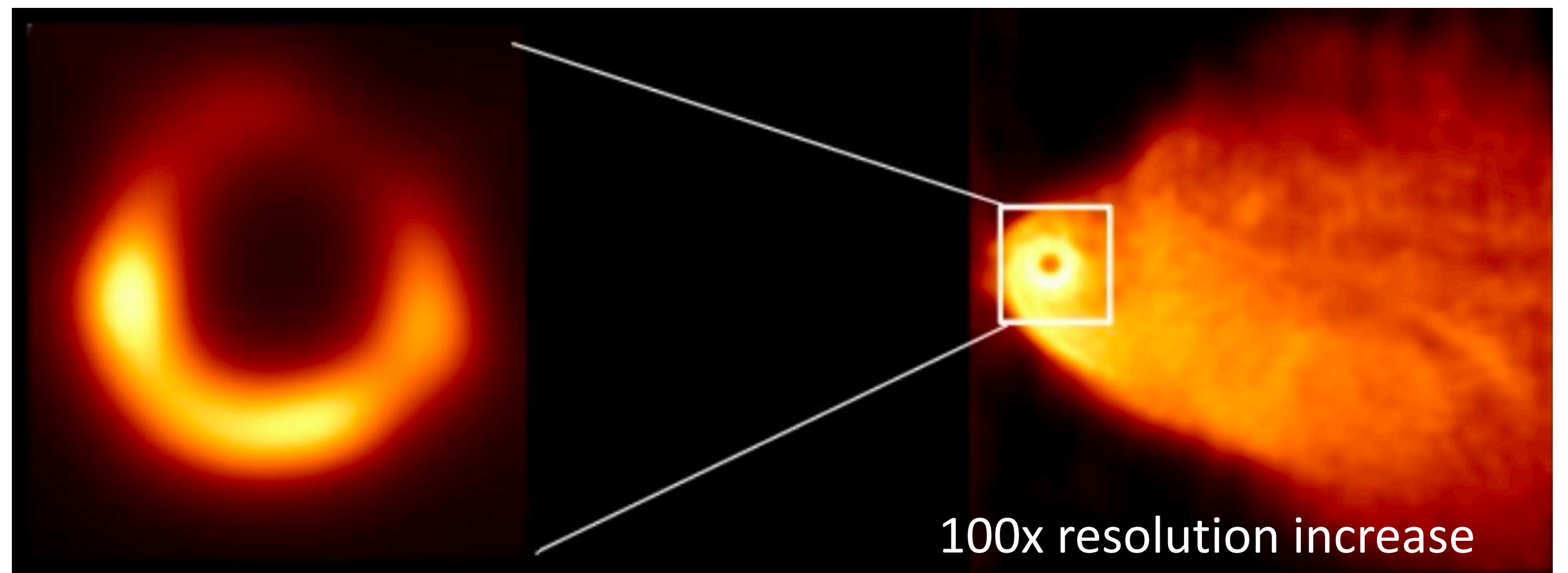
---

Derivatives are the most costly and difficult to use algorithms

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



With Enzyme differentiation:  
1 hour on 1 thread



# Existing AD Approaches (1/3)

---

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

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

Manually  
Rewrite



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

## Existing AD Approaches (2/3)

---

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

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

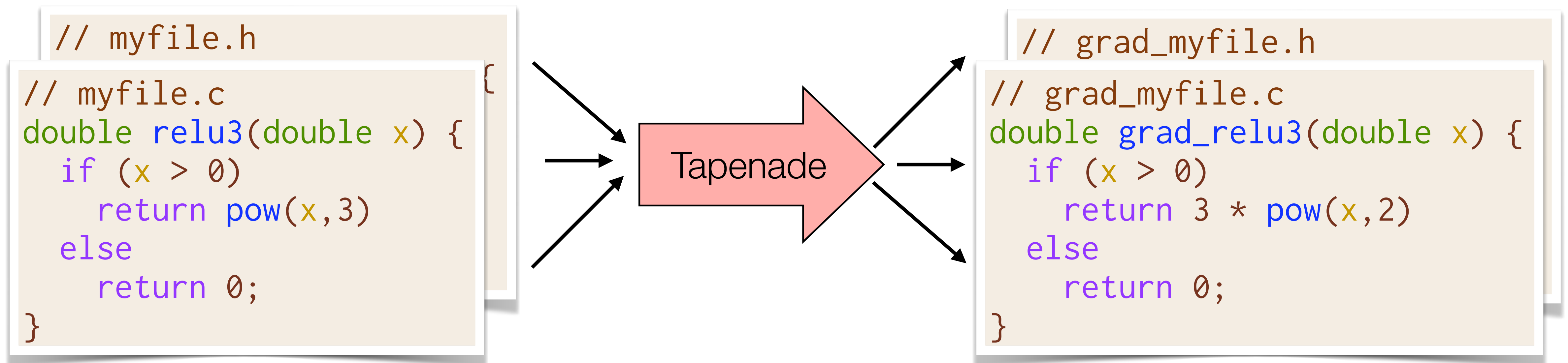
```
adept::Stack stack;
adept::adouble inp = 3.14;

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

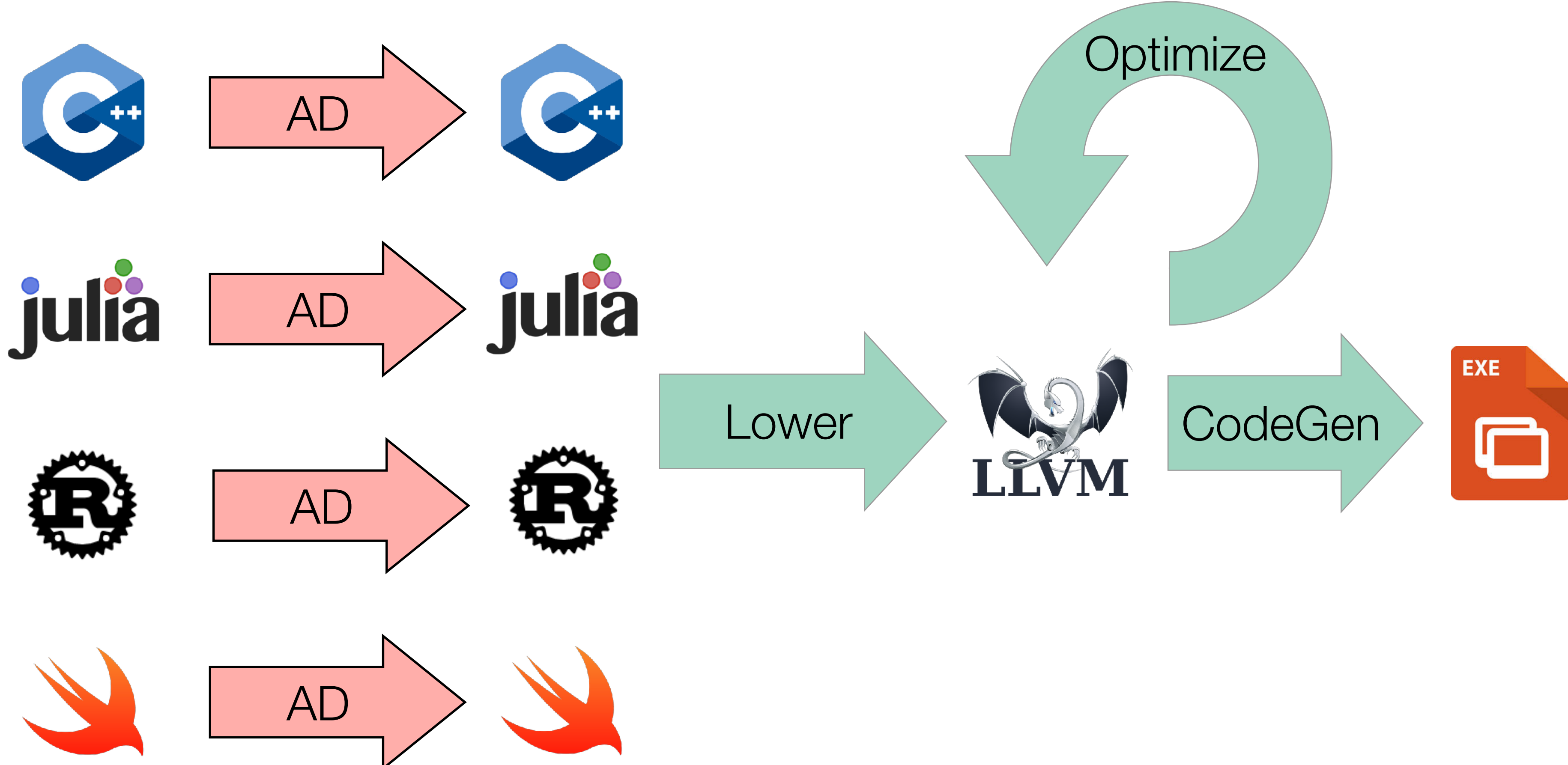
// Interpret all stack instructions
double res = inp.get_gradient(3.14);
```

# Existing AD Approaches (3/3)

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



# Existing Automatic Differentiation Pipelines



# Case Study: Vector Normalization

---

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

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

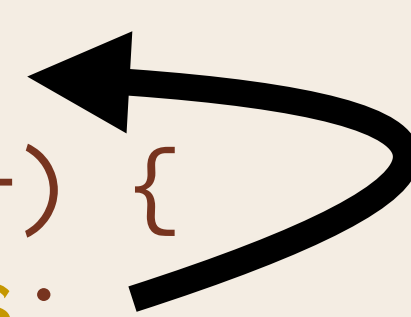
    for (int i=0; i<n; i++) {
        out[i] = in[i] / mag(in);
    }
}
```

# Case Study: Vector Normalization

---

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

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



# Optimization & Automatic Differentiation

$O(n^2)$

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

Optimize

$O(n)$

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

AD

$O(n)$

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

# Optimization & Automatic Differentiation

$O(n^2)$

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

Optimize

$O(n)$

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

AD

$O(n)$

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

$O(n^2)$

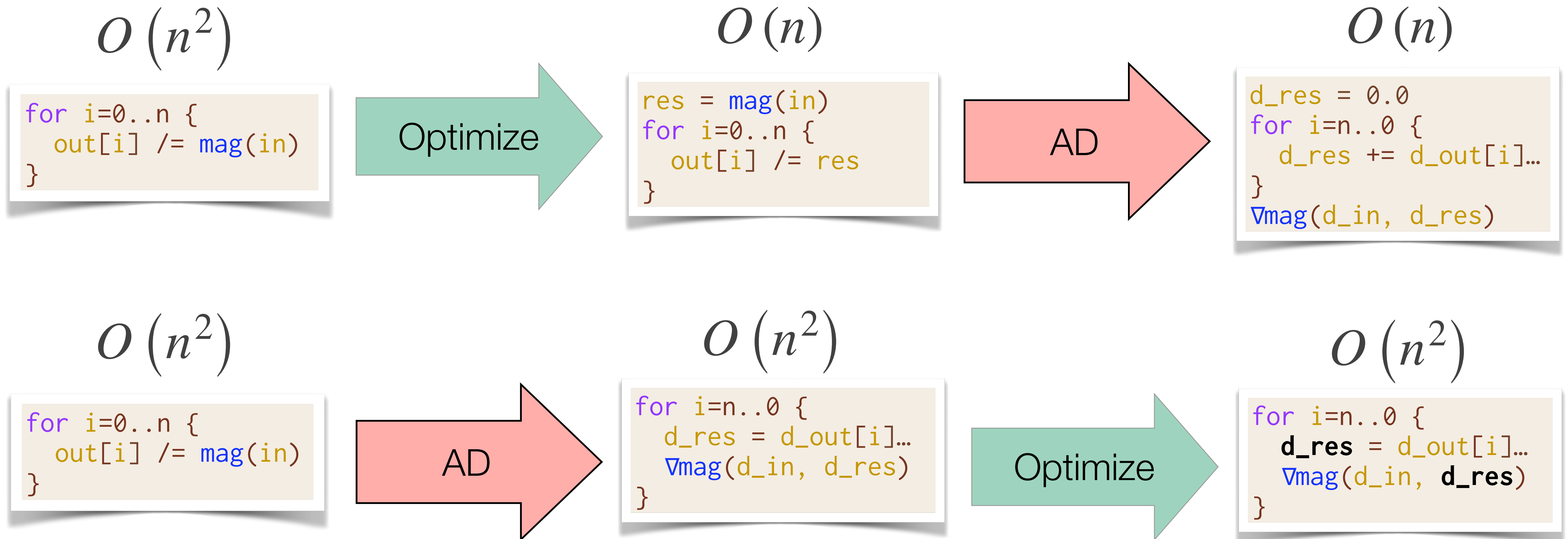
```
for i=0..n {  
  out[i] /= mag(in)  
}
```

AD

$O(n^2)$

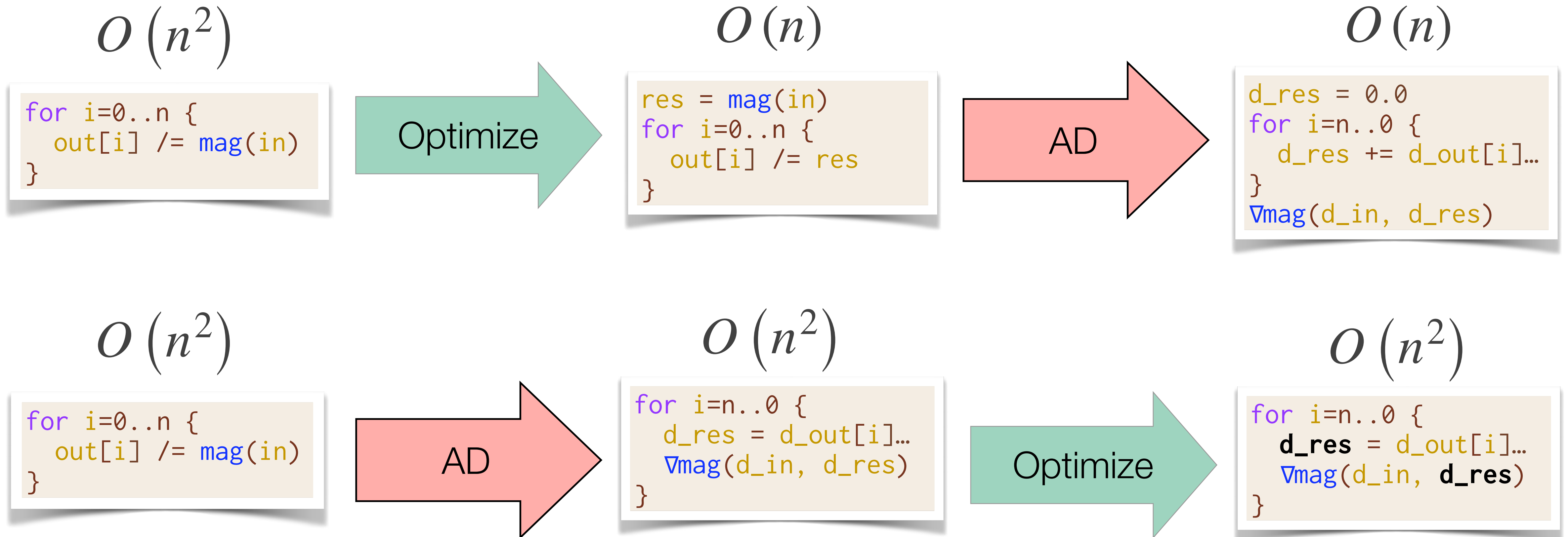
```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

# Optimization & Automatic Differentiation



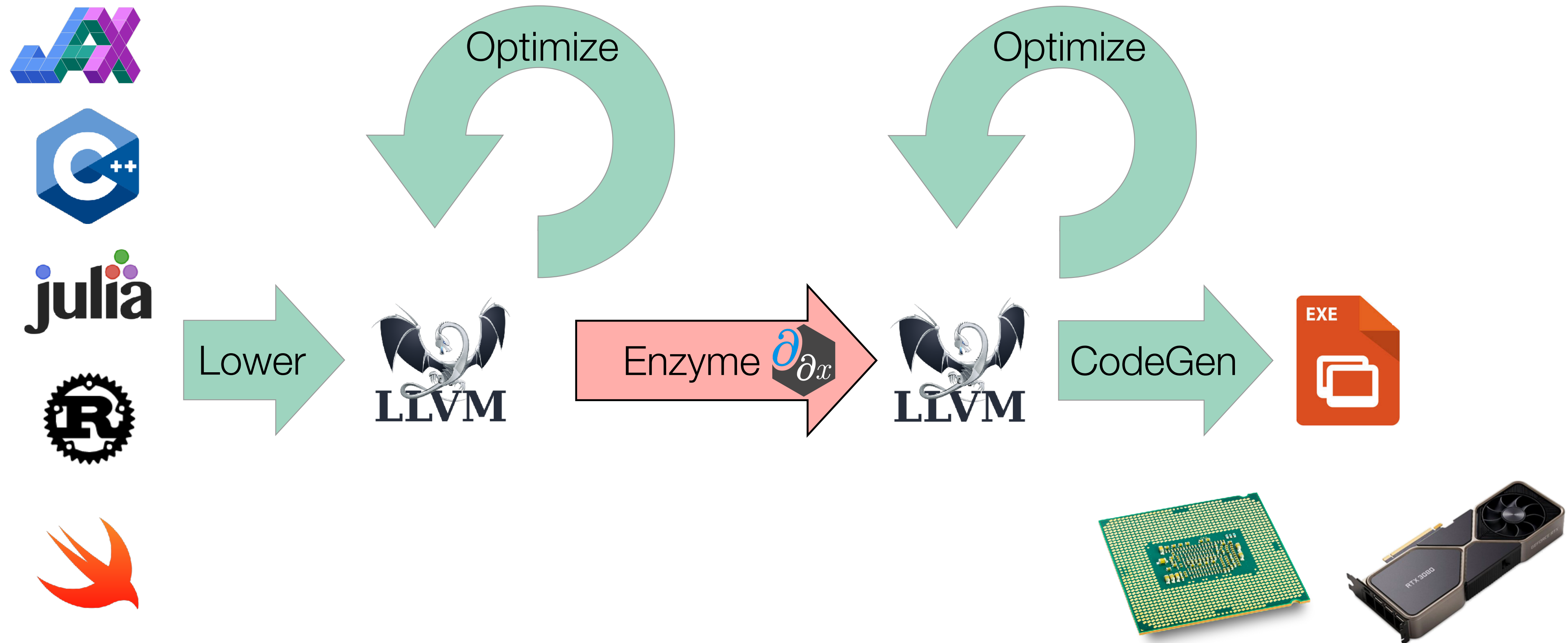
# Optimization & Automatic Differentiation

Differentiating after optimization can create *asymptotically faster* gradients!



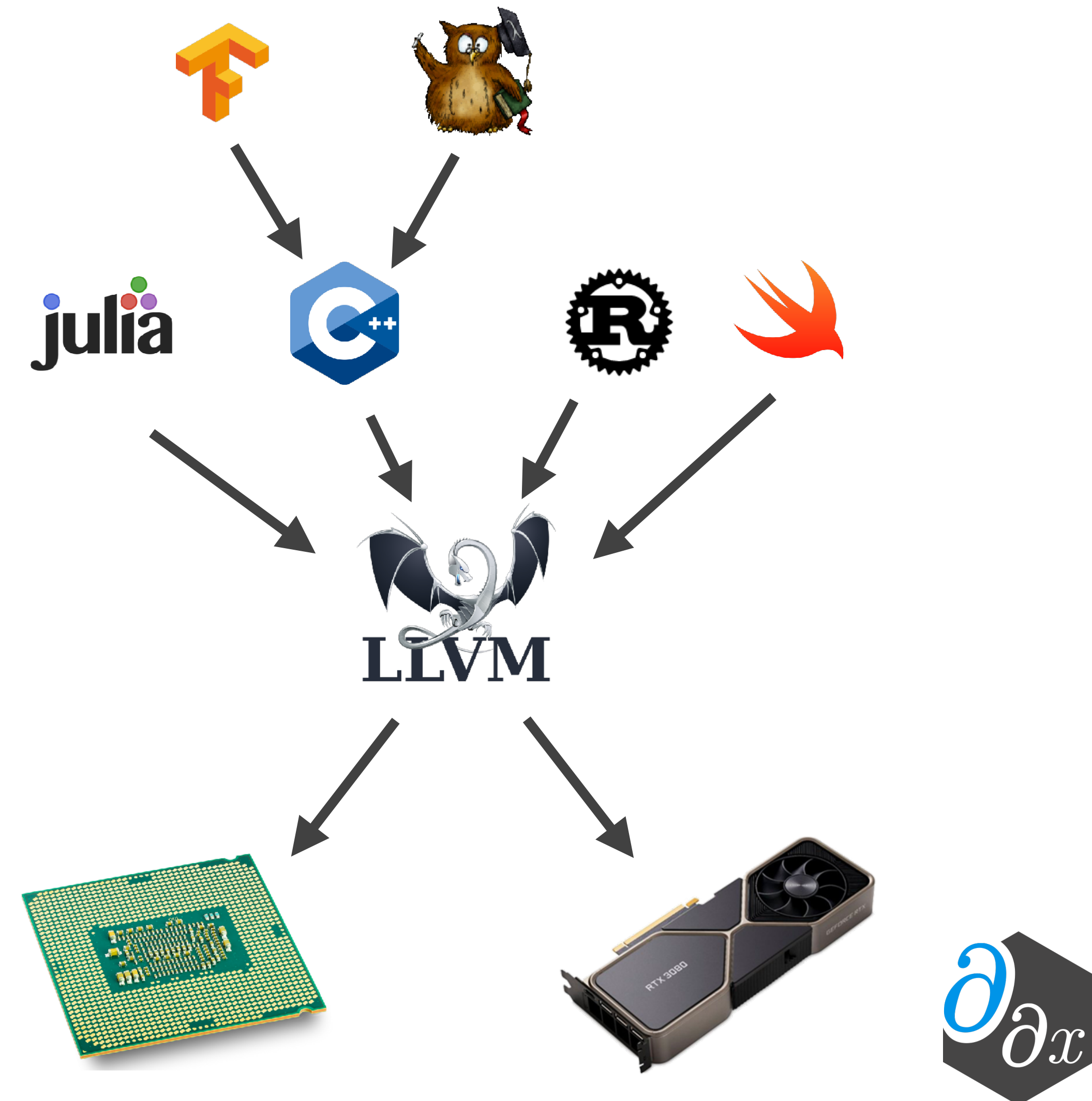
# Enzyme Approach

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



# Why Does Enzyme Use LLVM?

- Generic low-level compiler infrastructure with many frontends
  - “Cross platform assembly”
  - Many backends (CPU, CUDA, AMDGPU, etc)
- Well-defined semantics
- Large collection of optimizations and analyses



# Case Study: ReLU3

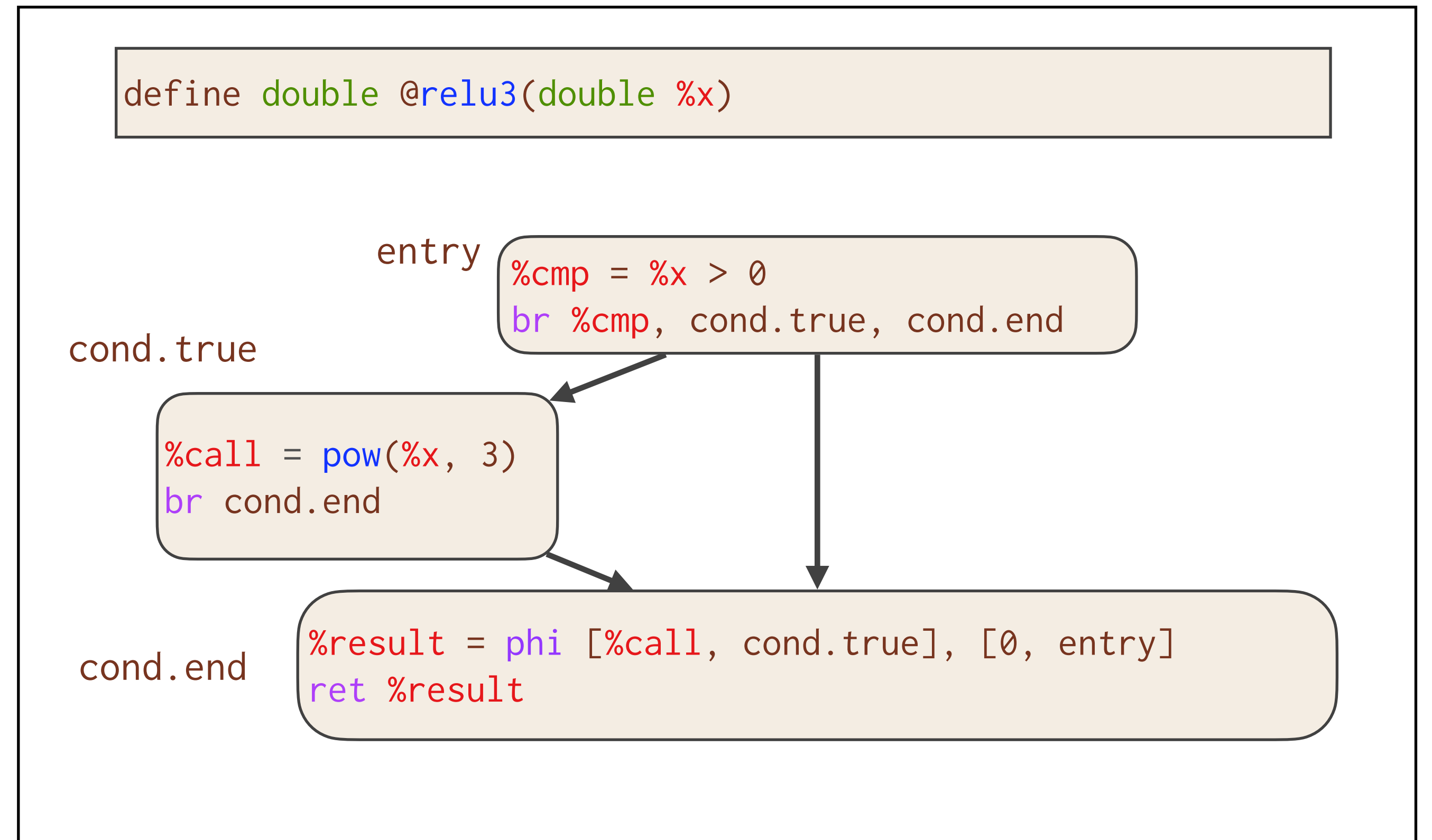
## C Source

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

## Enzyme Usage

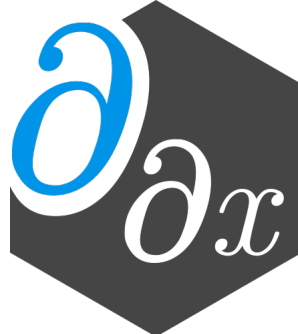
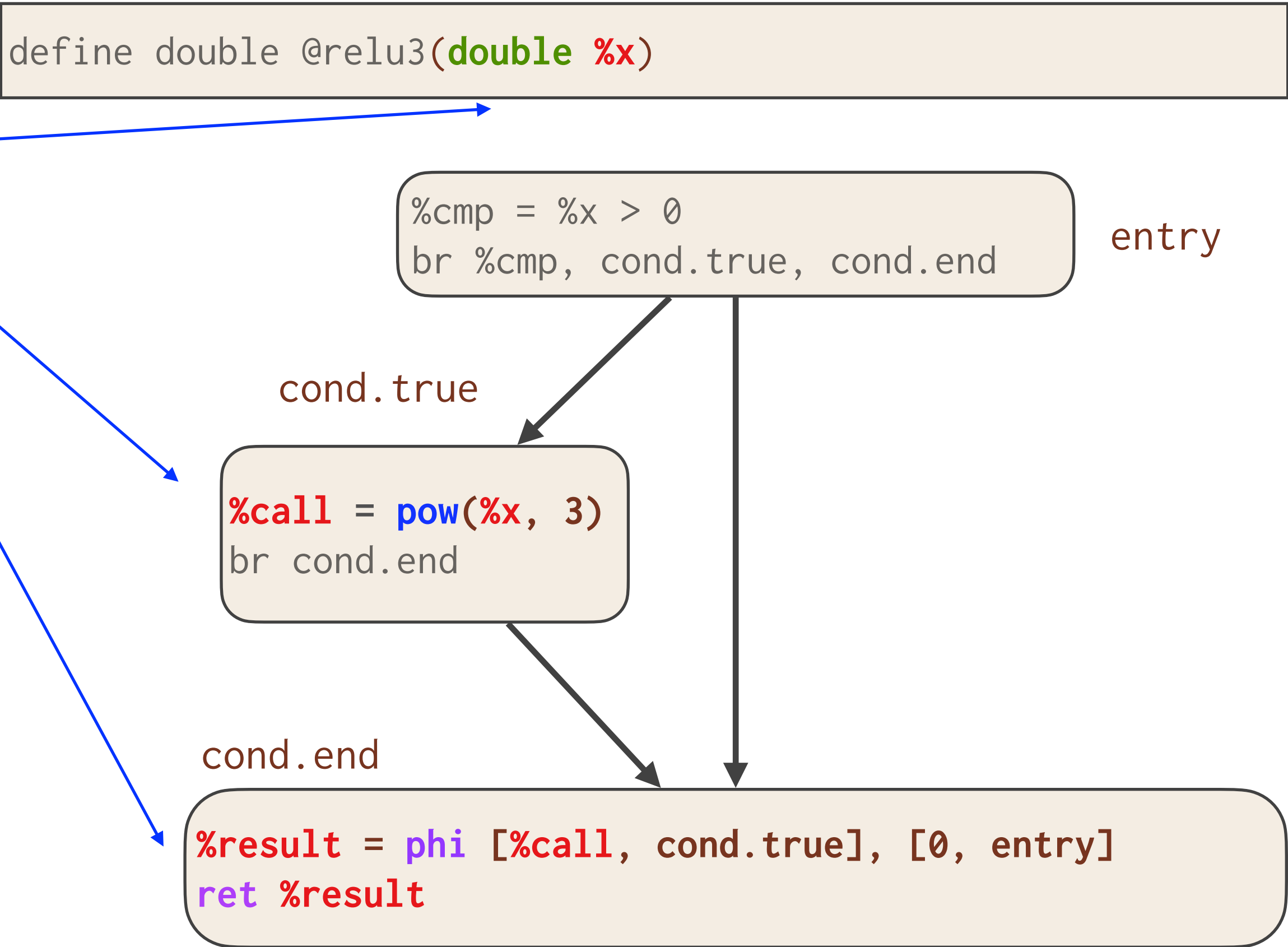
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

## LLVM

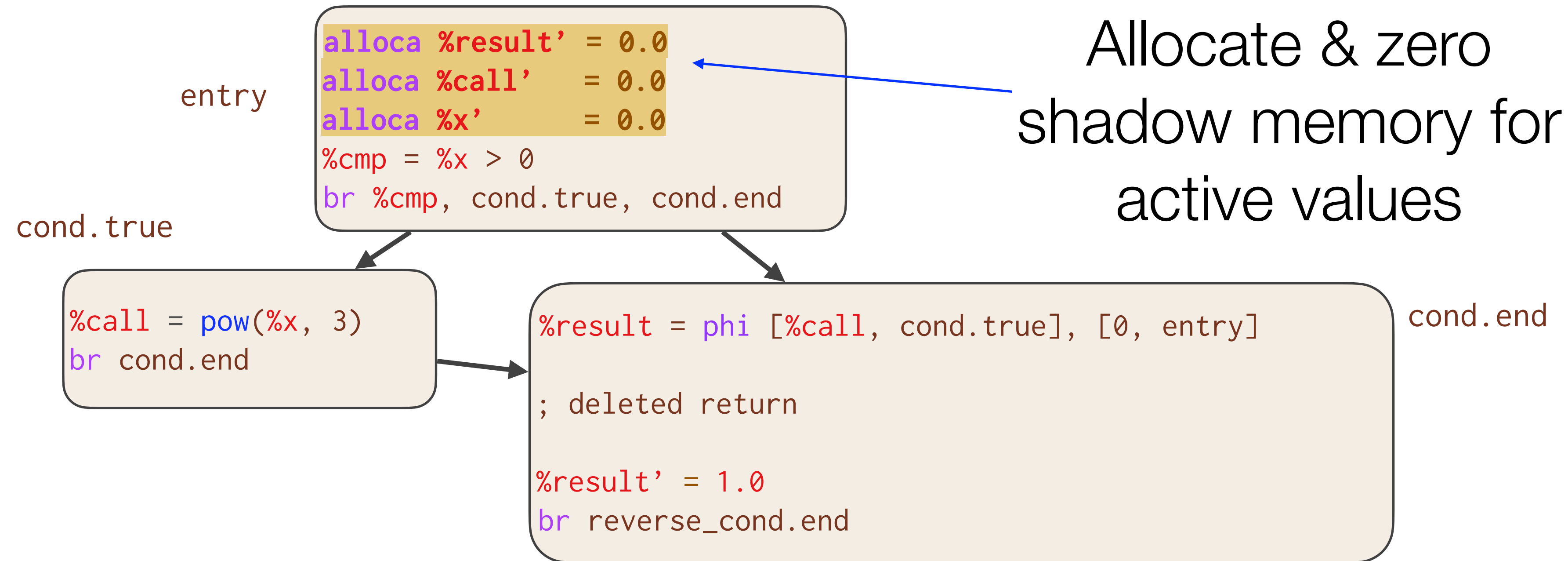


# Case Study: ReLU3

Active Instructions

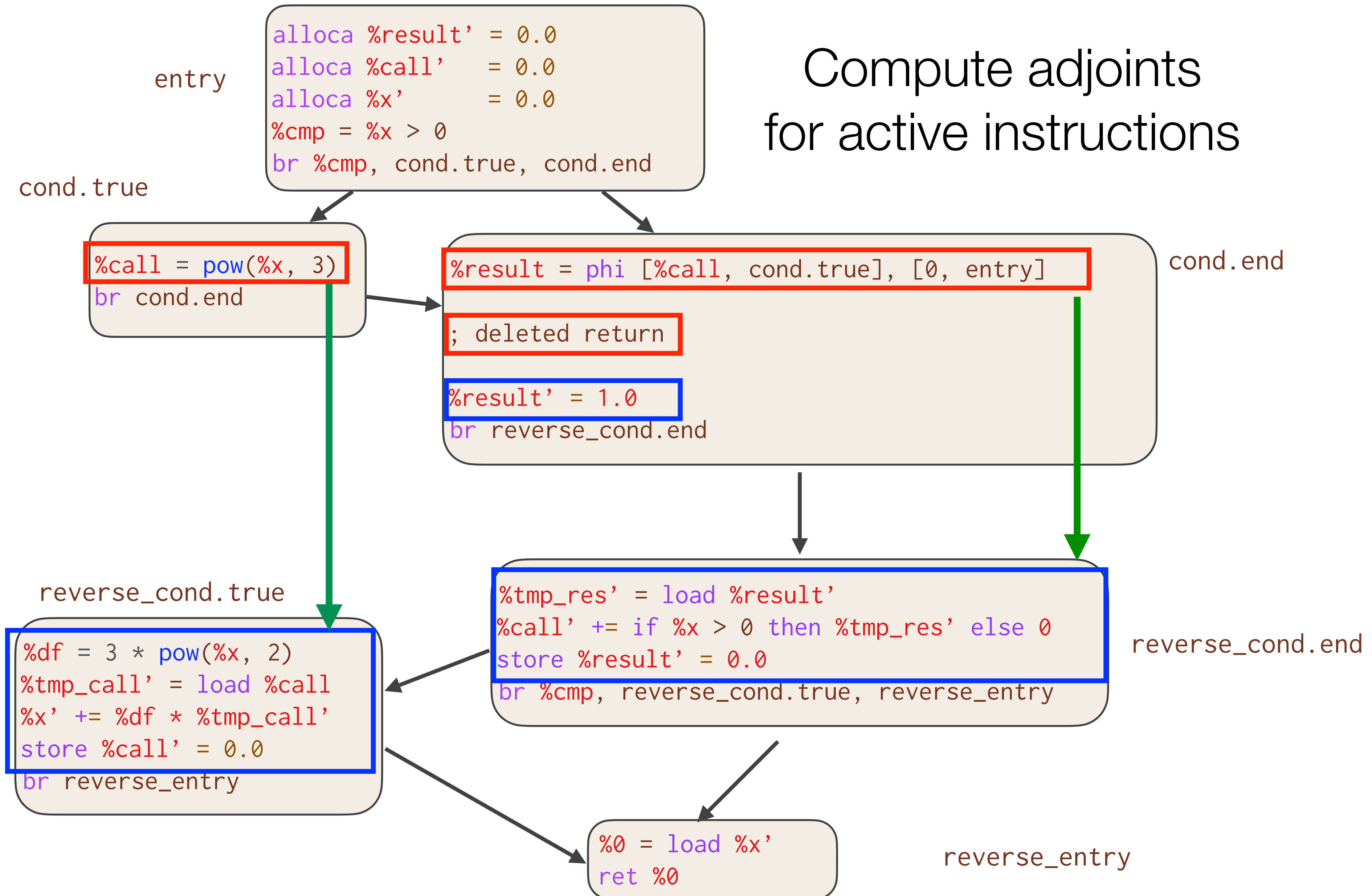


```
define double @diffe_relu3(double %x, double %differet)
```



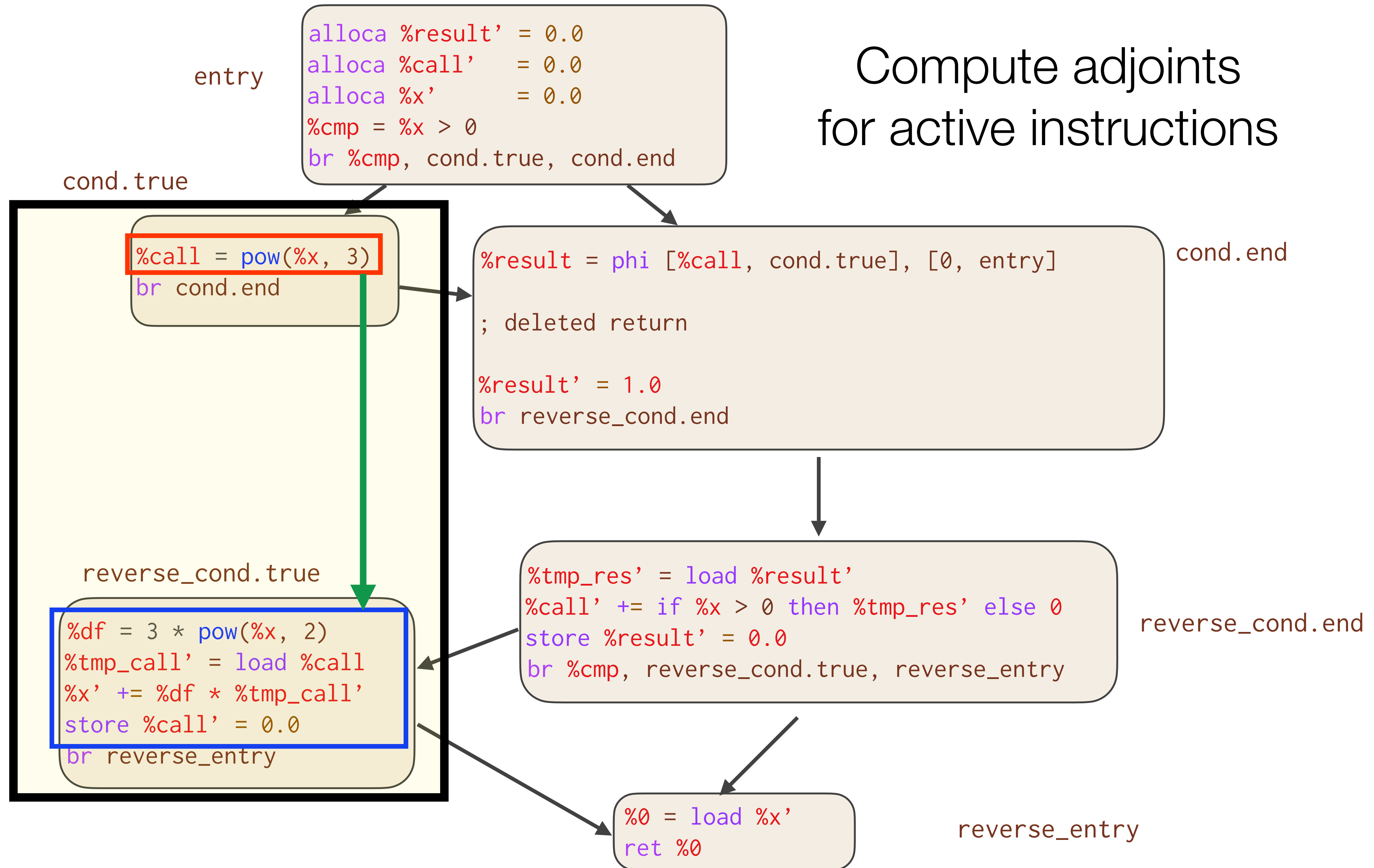
```
define double @diffe_relu3(double %x, double %differet)
```

## Compute adjoints for active instructions



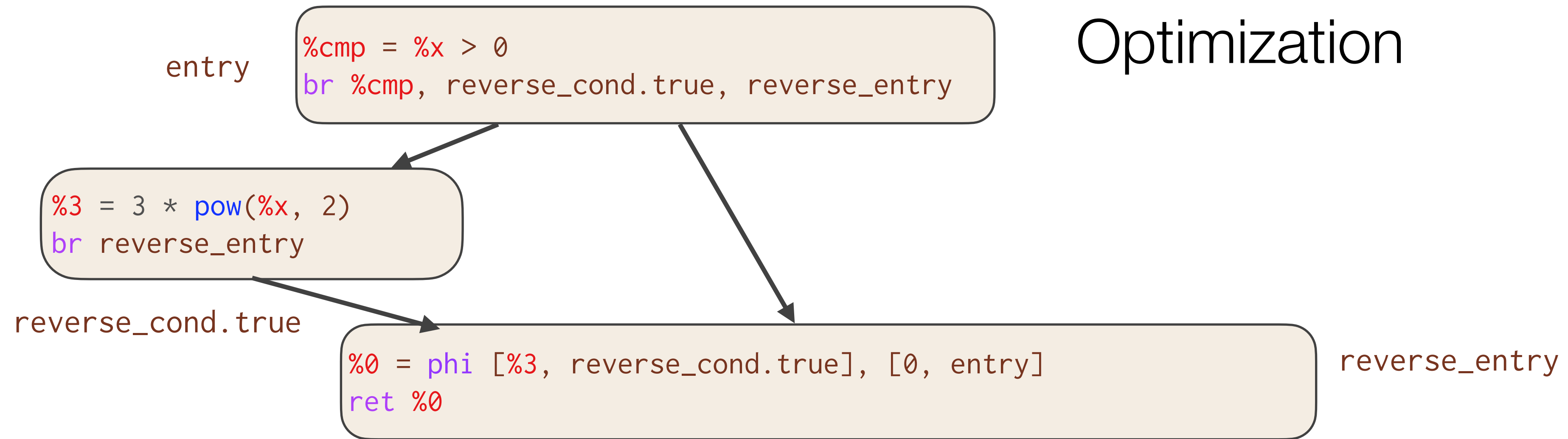
```
define double @diffe_relu3(double %x, double %differet)
```

# Compute adjoints for active instructions



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

## Post Optimization

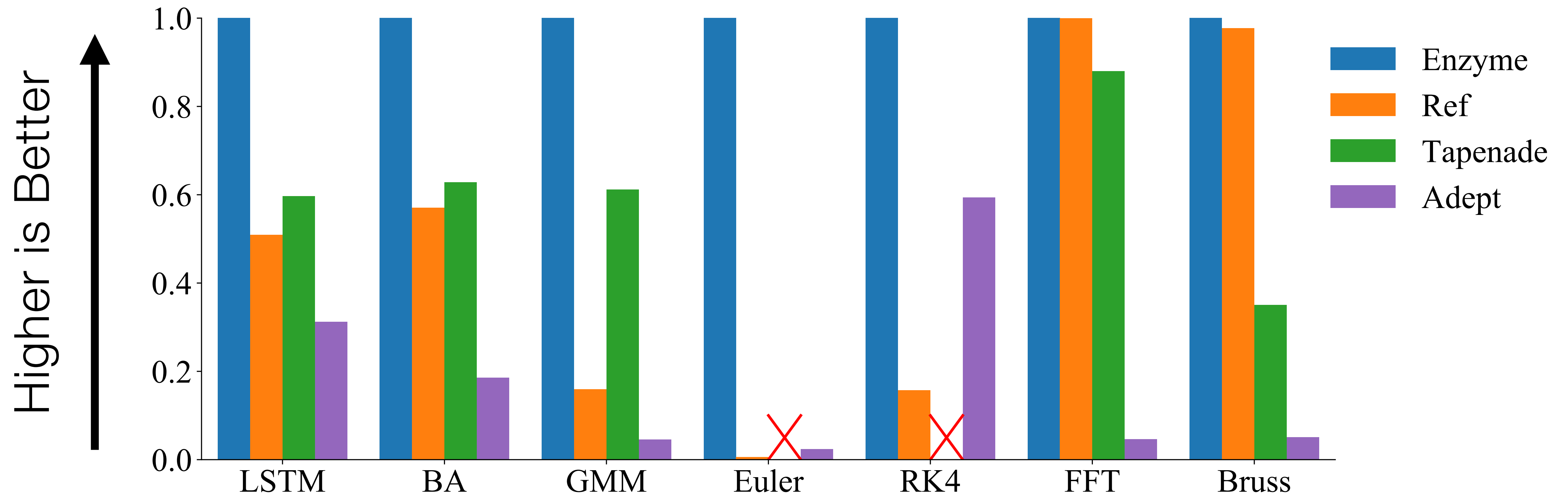


Essentially the optimal hand-written gradient!

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



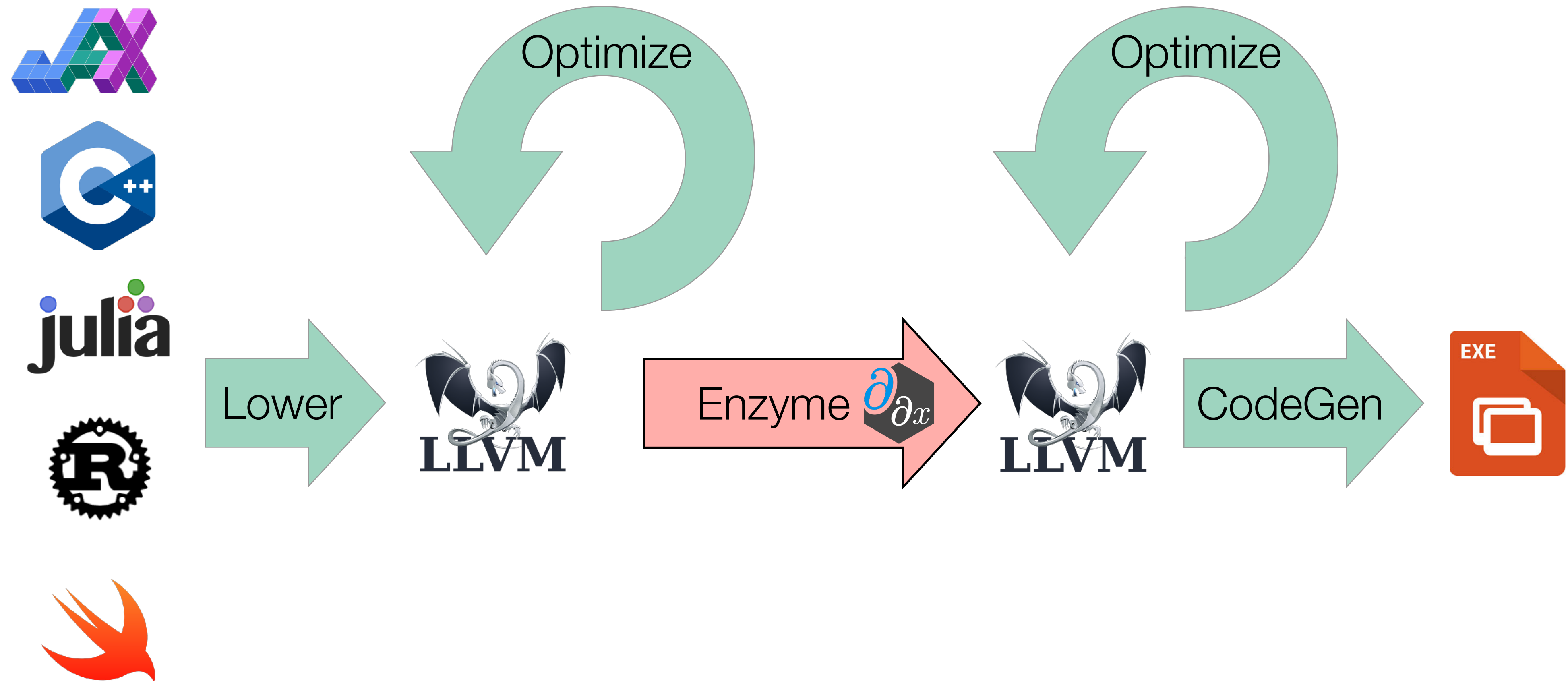
# Enzyme CPU Speedups [NeurIPS'20]



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

# Enzyme Approach

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



# Automatic Differentiation & GPUs [MCPHNSD @ SC'21]

---

- Prior work has not explored reverse mode AD of existing GPU kernels
  1. Reversing parallel control flow can lead to incorrect results
  2. Complex performance characteristics make it difficult to synthesize efficient code
  3. Resource limitations can prevent kernels from running at all



# Challenges of Parallel AD

- The adjoint of an instruction increments the derivative of its input
- Benign read race in forward pass => Write race in reverse pass (undefined behavior)

```
void set(double* ar, double val) {  
    parallel_for(int i=0; i<10; i++)  
        ar[i] = val;  
}
```

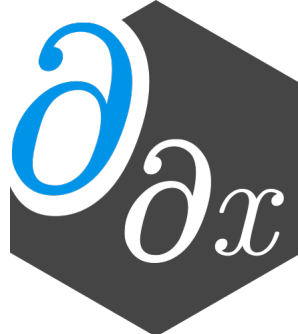
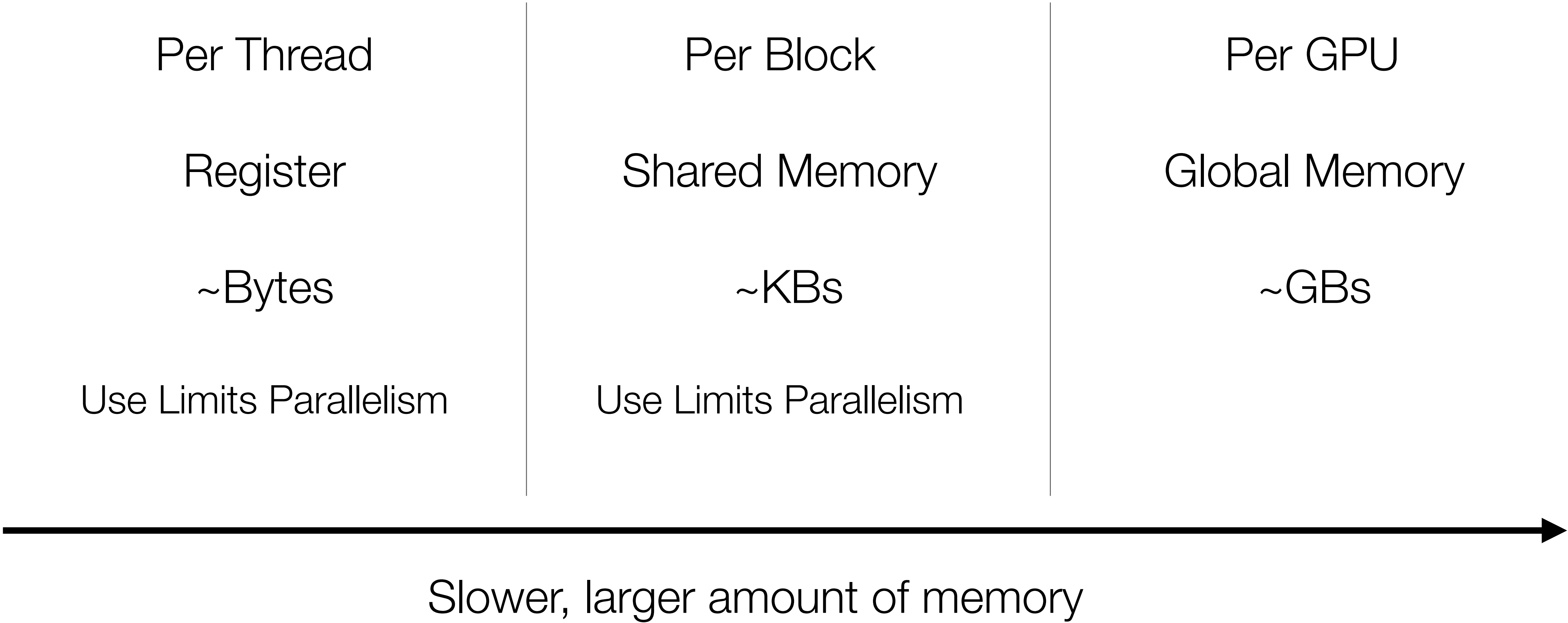
Read Race

```
double gradient_set(double* ar, double* d_ar,  
                  double val) {  
    double d_val = 0.0;  
  
    parallel_for(int i=0; i<10; i++)  
        ar[i] = val;  
  
    parallel_for(int i=0; i<10; i++) {  
        d_val += d_ar[i];  
        d_ar[i] = 0.0;  
    }  
  
    return d_val;  
}
```

Write Race

# GPU Memory Hierarchy

---



# Correct and Efficient Derivative Accumulation

Thread-local memory

- Non-atomic load/store

```
__device__  
void f(...) {  
  
    // Thread-local var  
    double y;  
  
    ...  
  
    d_y += val;  
}
```

Same memory location across all threads (some shared mem)

- Parallel Reduction

```
// Same var for all threads  
double y;  
  
__device__  
void f(...) {  
  
    ...  
  
    reduce_add(&d_y, val);  
}
```

Others [always legal fallback]

- Atomic increment

```
__device__  
// Unknown thread-aliasing  
void f(double* y) {  
  
    ...  
  
    atomic { d_y += val; }  
}
```

Slower



# Synchronization Primitives

---

- Synchronization (`sync_threads`) ensures all threads finish executing `codeA` before executing `codeB`
- Sync is only necessary if A and B may access to the same memory
- Assuming the original program is race-free, performing a sync at the corresponding location in the reverse ensures correctness
- Prove correctness of algorithm by cases

```
codeA();  
sync_threads;  
codeB();
```

# Case 1: Store, Sync, Load

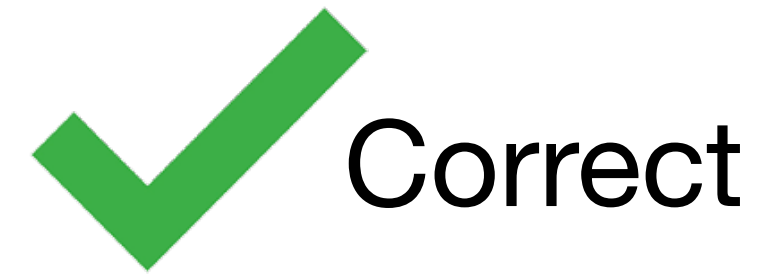
---

```
codeA(); // store %ptr
sync_threads;

codeB(); // load %ptr
...

diffe_codeB(); // atomicAdd %d_ptr
sync_threads;

diffe_codeA(); // load %d_ptr
                // store %d_ptr = 0
```



- Load of `d_ptr` must happen after all `atomicAdds` have completed

# CUDA Example

```
__device__
void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__device__
void __enzyme_autodiff(void*, ...);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {
    __enzyme_autodiff((void*)inner,
                     a, da, x, dx, y, dy);
}
```

```
__device__
void diffe_inner(float* a, float* da,
                float* x, float* dx,
                float* y, float* dy) {
    // Forward Pass

    y[threadIdx.x] = a[0] * x[threadIdx.x];

    // Reverse Pass

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    atomic { dx[threadIdx.x] += dx_tmp; }

    float da_tmp = x[threadIdx.x] * dy;
    atomic { da[0] += da_tmp; }
}
```



# CUDA Example

```
__device__
void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__device__
void __enzyme_autodiff(void*, ...);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {
    __enzyme_autodiff((void*)inner,
                     a, da, x, dx, y, dy);
}
```

```
__device__
void diffe_inner(float* a, float* da,
                 float* x, float* dx,
                 float* y, float* dy) {
    // Forward Pass

    y[threadIdx.x] = a[0] * x[threadIdx.x];

    // Reverse Pass

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    dx[threadIdx.x] += dx_tmp;

    float da_tmp = x[threadIdx.x] * dy;
    reduce_accumulate(&da[0], da_tmp);
}
```



# CUDA.jl / AMDGPU.jl Example

```
function compute!(inp, out)
    s_D = @cuStaticSharedMem eltype(inp) (10, 10)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@cuda grad_compute!(Duplicated(inp, d_inp),
                   Duplicated(out, d_out))
```

```
function compute!(inp, out)
    s_D = AMDGPU.alloc_special(...)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@rocm grad_compute!(Duplicated(inp, d_inp),
                   Duplicated(out, d_out))
```

See Below For Full Code Examples

<https://github.com/wsmoses/Enzyme-GPU-Tests/blob/main/DG/>



# Efficient GPU Code

---

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

```
// Forward Pass
out[i] = x[i] * x[i];
x[i] = 0.0f;

// Reverse (gradient) Pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

# Efficient Correct GPU Code

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

```
double* x_cache = new double[...];

// Forward Pass

out[i] = x[i] * x[i];
x_cache[i] = x[i];

x[i] = 0.0f;

// Reverse (gradient) Pass

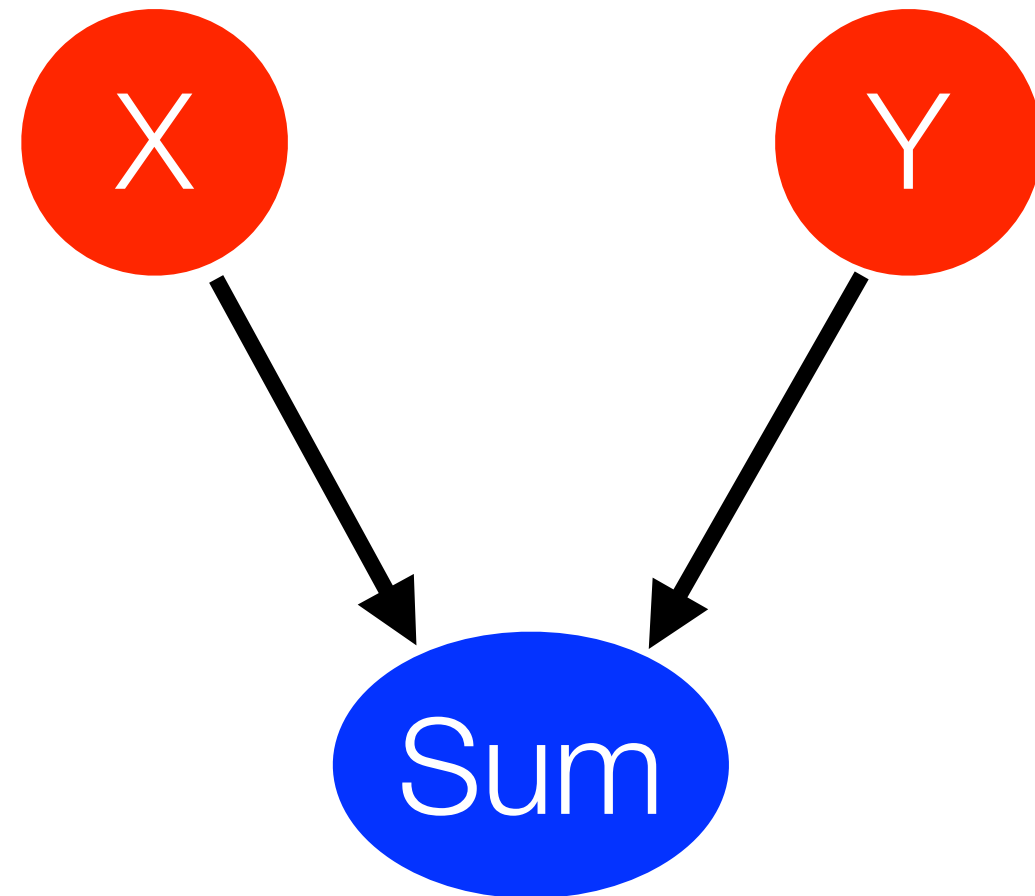
...
grad_x[i] += 2 * x_cache[i]
             * grad_out[i];
...

delete[] x_cache;
```

# Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.

Overwritten:

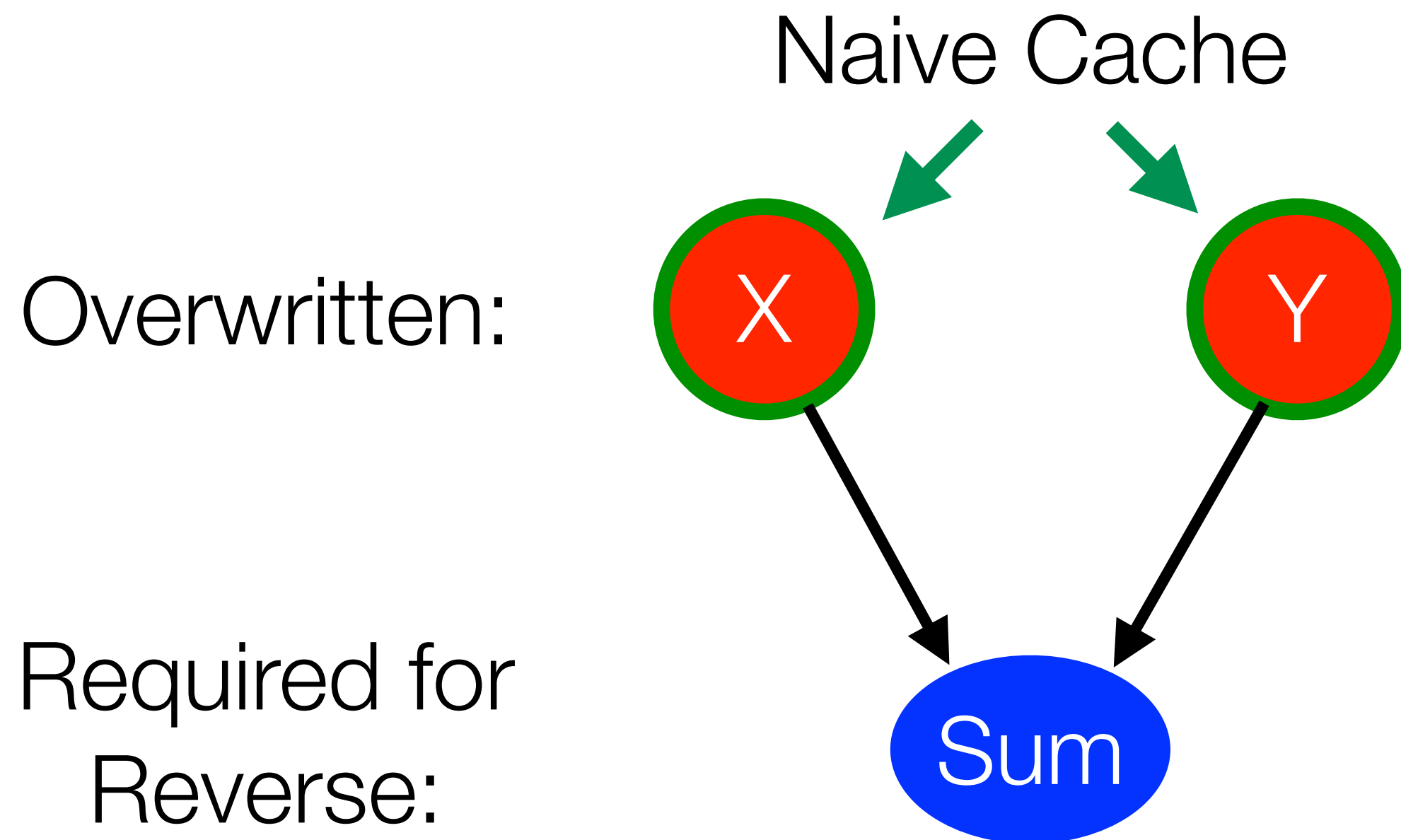


Required for  
Reverse:

```
for(int i=0; i<10; i++) {  
    double sum = x[i] + y[i];  
  
    use(sum);  
}  
  
overwrite(x, y);  
grad_overwrite(x, y);  
  
for(int i=9; i>=0; i--) {  
    ...  
    grad_use(sum);  
}
```

# Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.



```
double* x_cache = new double[10];
double* y_cache = new double[10];

for(int i=0; i<10; i++) {
    double sum = x[i] + y[i];
    x_cache[i] = x[i];
    y_cache[i] = y[i];
    use(sum);
}

overwrite(x, y);
grad_overwrite(x, y);

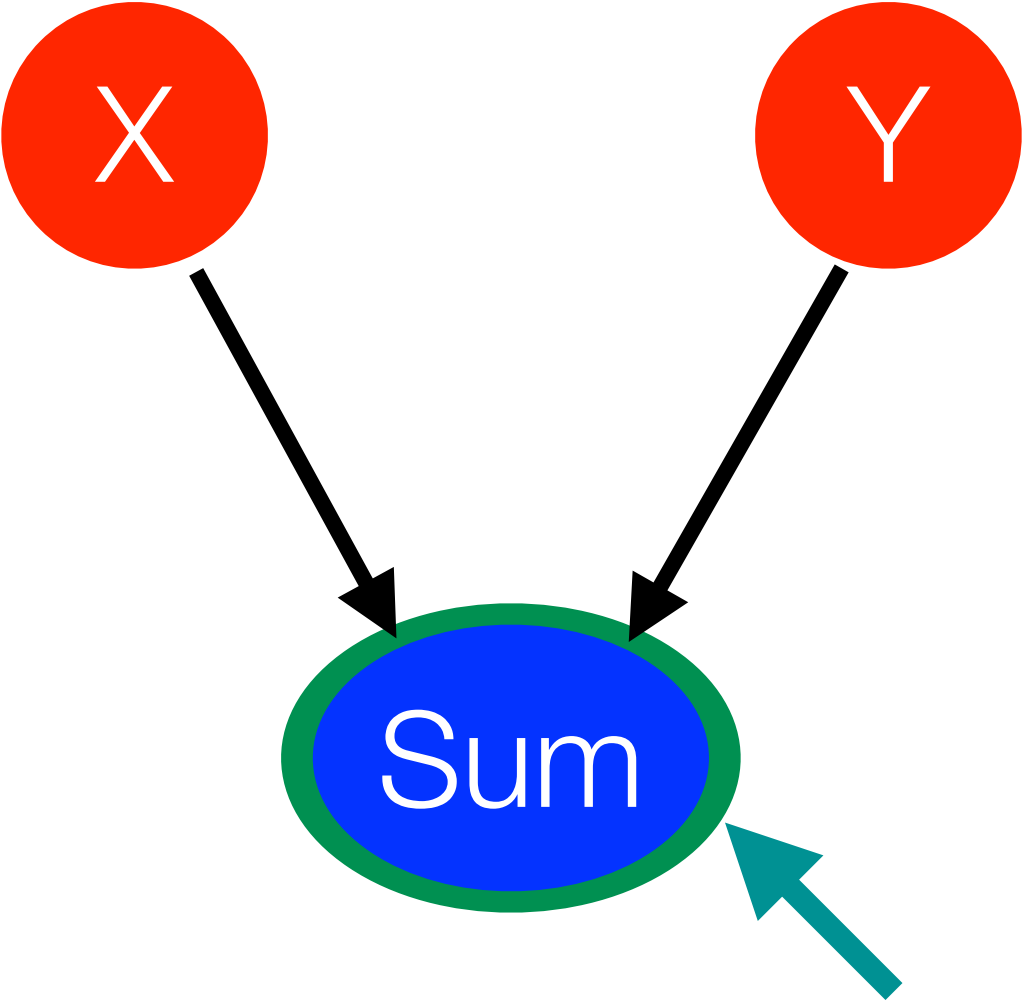
for(int i=9; i>=0; i--) {
    double sum = x_cache[i] + y_cache[i];
    grad_use(sum);
}
```

# Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.

```
double* sum_cache = new double[10];  
  
for(int i=0; i<10; i++) {  
    double sum = x[i] + y[i];  
    sum_cache[i] = sum;  
  
    use(sum);  
}  
  
overwrite(x, y);  
grad_overwrite(x, y);  
  
for(int i=9; i>=0; i--) {  
    grad_use(sum_cache[i]);  
}
```

Overwritten:



Required for Reverse:

Smallest Cache

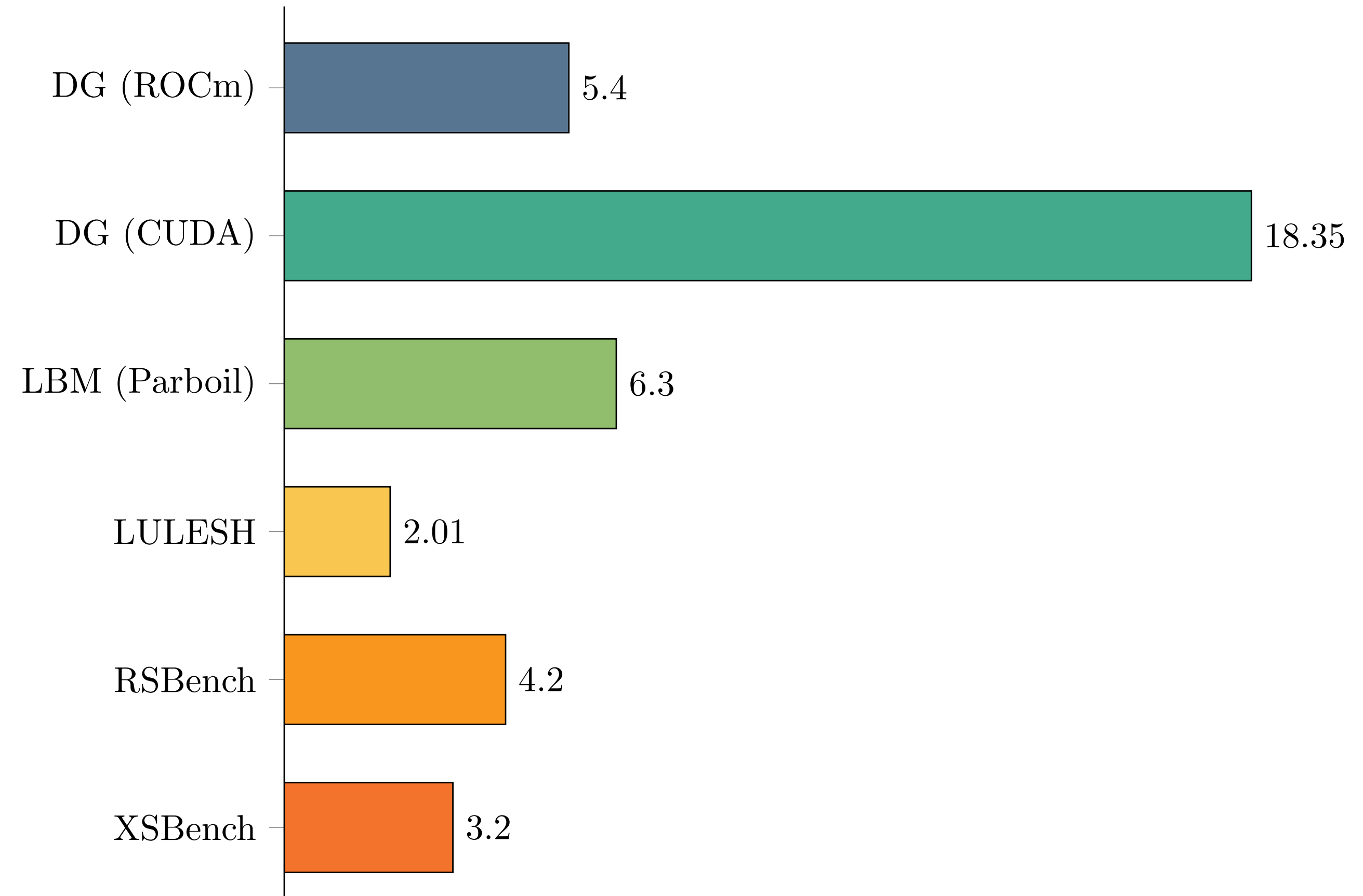
# Novel AD + GPU Optimizations

---

- See our SC'21 paper for more (<https://c.wsmoses.com/papers/EnzymeGPU.pdf>)  
Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. SC, 2021
- [AD] Cache LICM/CSE
- [AD] Min-Cut Cache Reduction
- [AD] Cache Forwarding
- [GPU] Merge Allocations
- [GPU] Heap-to-stack (and register)
- [GPU] Alias Analysis Properties of SyncThreads
- ...

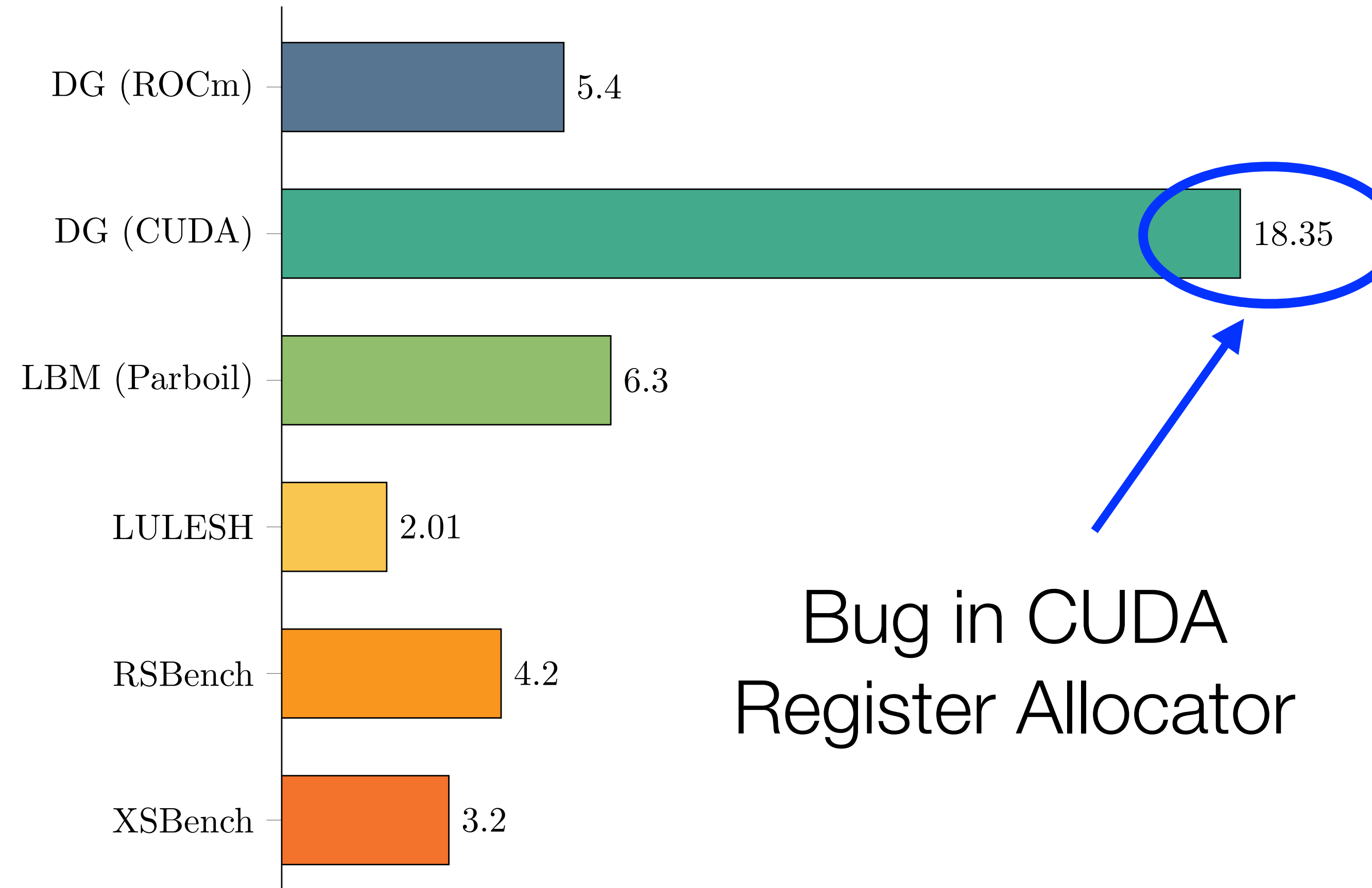
# GPU Gradient Overhead [MCPHNMJ'21]

- Evaluation of both original code and gradient
  - DG: Discontinuous-Galerkin integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute bound, respectively)



# GPU Gradient Overhead [MCPHNMJ'21]

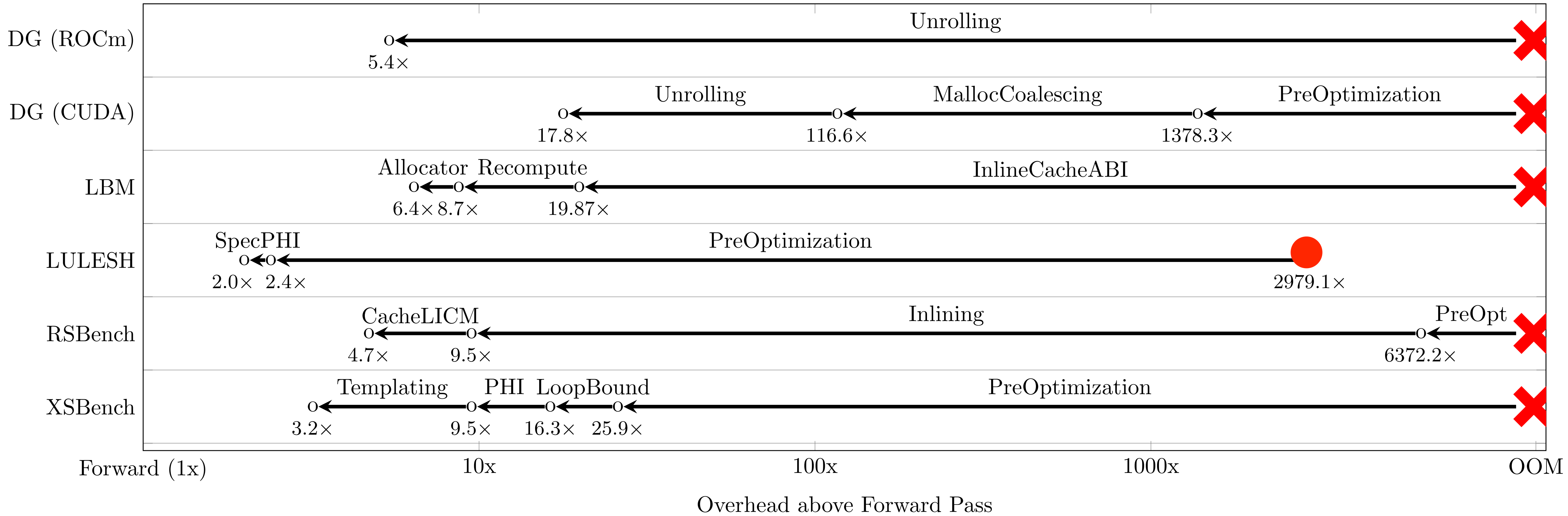
- Evaluation of both original code and gradient
  - DG: Discontinuous-Galerkin integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute bound, respectively)



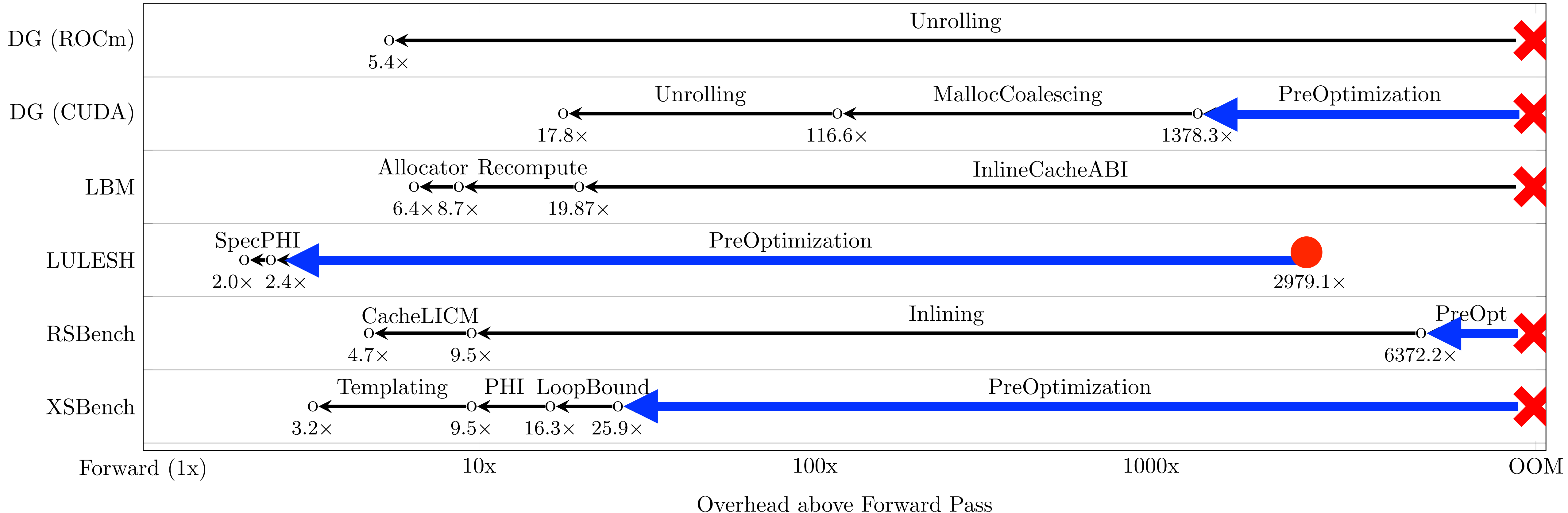
Bug in CUDA  
Register Allocator



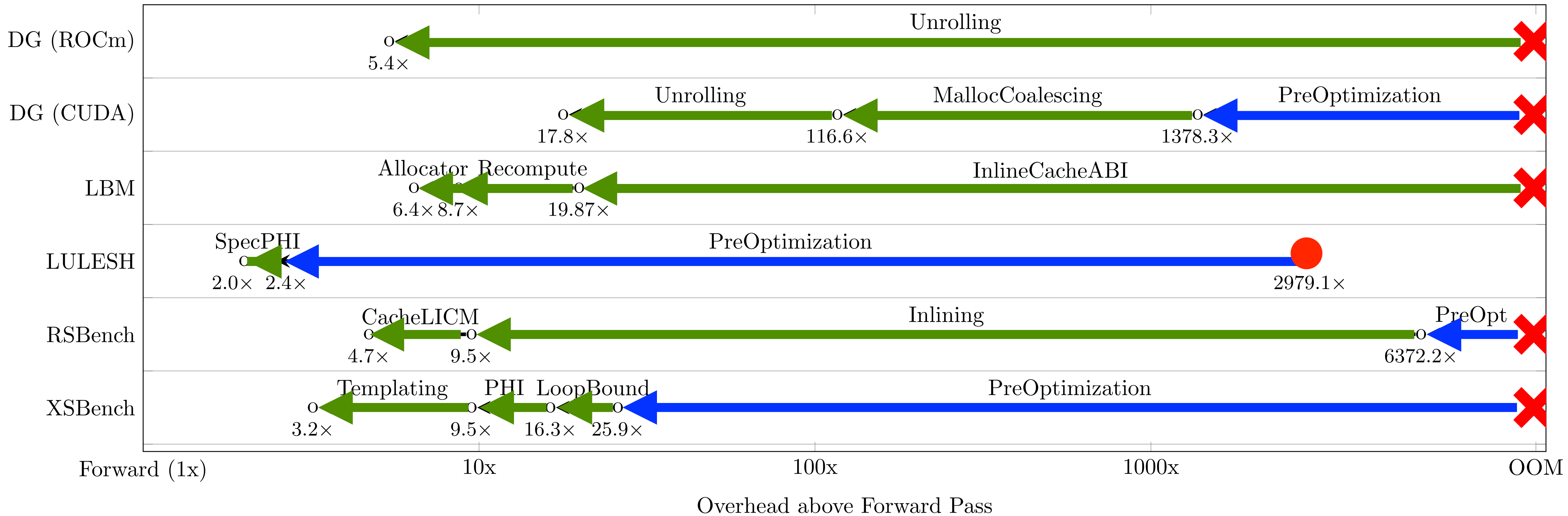
# Ablation Analysis of Optimizations



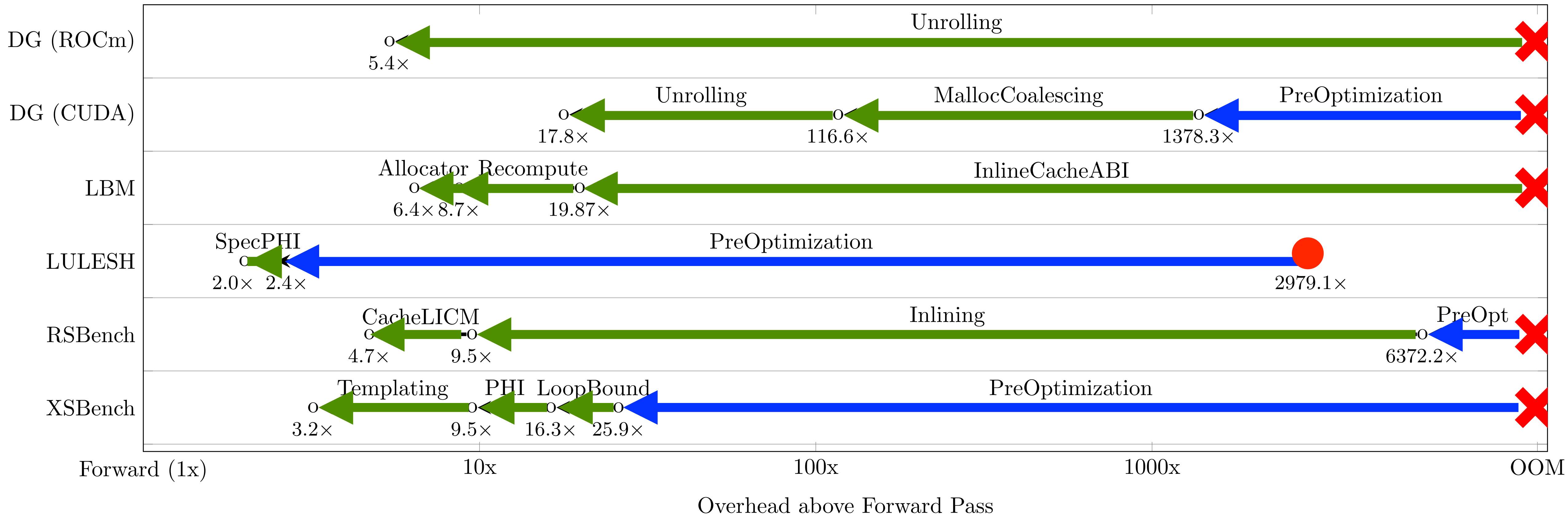
# Ablation Analysis of Optimizations



# Ablation Analysis of Optimizations



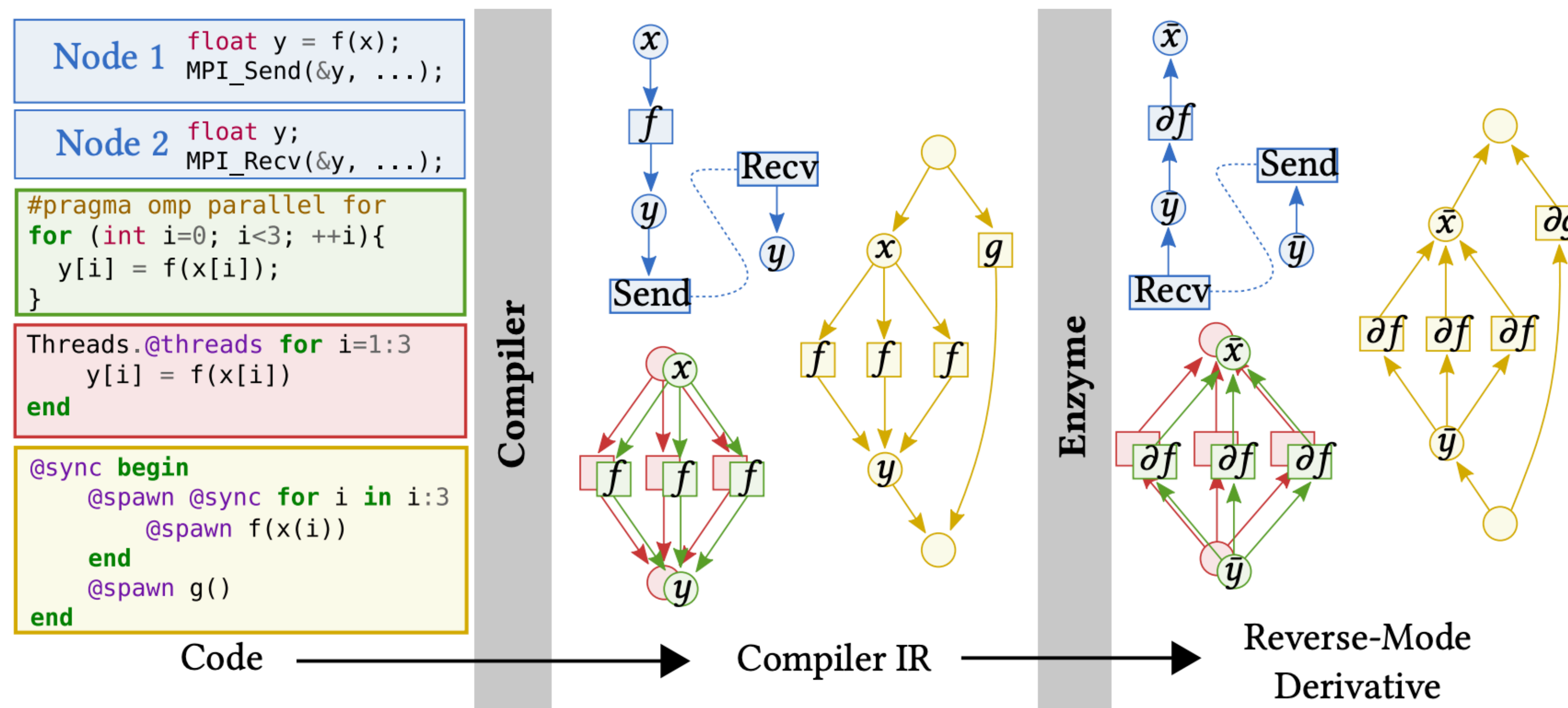
# Ablation Analysis of Optimizations



GPU AD is Intractable Without Optimization!

# Common Framework for Parallel AD [SC'22]

- Common infrastructure for supporting parallel AD (caching, race-resolution, gradient accumulation) enables parallel differentiation independent of framework or language.



- Enables differentiation of a combination of GPU (e.g. CUDA, ROCm), CPU (OpenMP, Julia Tasks, RAJA), Distributed (MPI, MPI.jl), and more



# 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

```
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
```

# 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}
```

```

julia> @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}

```

# 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)

```
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

$x + 0 \rightarrow x$

`transpose(transpose(x)) -> x`

`transpose(matmul(a,b)) ->  
matmul(b, a)`

Often require program context

`transpose(convert(reshape(x)))  
<-> reshape(convert(transpose(x)))`

`slice(add(a, b)) ->  
add(slice(a), slice(b))`

`mul(pad(x, 0), y) ->  
pad(mul(x, 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
```

# How Does Raising & Tensor Transformations Impact AD?

### Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme

William S. Moses<sup>1</sup>, Valentin Churavy<sup>1</sup>, Ludger Paehler<sup>1</sup>, Jan Hückelheim<sup>1</sup>, Sri Hari Krishna Narayanan<sup>1</sup>, Michel Schanen<sup>1</sup>, Johannes Doerfert<sup>1</sup>  
wsmoses@mit.edu, ludger.paehler@tum.de, jan.hueckelheim@tum.de, michel.schanen@doerfert.org, val@alum.mit.edu, Technical University of Munich<sup>1</sup>, Argonne National Laboratory<sup>2</sup>

**ABSTRACT**  
Computing derivatives is key to many algorithms in scientific computing and machine learning such as optimization, uncertainty quantification, and stability analysis. Enzyme is a LLVM compiler plugin that performs reverse-mode automatic differentiation (AD) and thus generates high-performance gradients of programs in languages including C++, Fortran, Julia, and Rust. Prior to this work, Enzyme and other AD tools were not capable of generating gradients of GPU kernels. Our paper presents a combination of novel techniques that make Enzyme the first fully automatic reverse-mode AD tool to generate gradients of GPU kernels. Since unlike other tools Enzyme performs automatic differentiation within a general-purpose compiler, we are able to introduce several novel GPU and AD-specific optimizations. To show the generality and efficiency of our approach, we compare gradients of GPU-based HPC applications, executed on NVIDIA and AMD GPUs. All benchmarks run within an order of magnitude of the original program's execution time. Without GPU and AD-specific optimizations, gradients of GPU kernels either fail to run from a lack of resources or have infeasible overhead. Finally, we demonstrate that increasing the problem size by either increasing the number of threads, increasing the work per thread, does not substantially impact overhead from differentiation.

**CCS CONCEPTS**  
• Mathematics of computing → Automatic differentiation  
• Software and its engineering → Source code generation  
Theory of computation → Parallel computing models; Shared memory algorithms; • Computing methodologies → Machine learning.

**KEYWORDS**  
Automatic Differentiation, AD, CUDA, ROCm, GPU, LLVM, HPC

**ACM Reference Format:**  
William S. Moses<sup>1</sup>, Valentin Churavy<sup>1</sup>, Ludger Paehler<sup>1</sup>, Jan Hückelheim<sup>1</sup>, Sri Hari Krishna Narayanan<sup>1</sup>, Michel Schanen<sup>1</sup>, Johannes Doerfert<sup>1</sup>, 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In: The International Conference for High Performance Computing, HPC '21, October 10–12, 2021, St. Louis, MO, USA. ACM, 2021. Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-1-4503-4211-1. DOI: <https://doi.org/10.1145/3470187>

```
void init(double* ar, int N, double val) {
    parallel_for(int i=0; i<N; i++)
        if (i % 1000000 == 0) ar[i] = val;
}

double gradient_init(double* ar, double* d_ar,
                    int N, double val) {
    double d_val = 0.0;
    parallel_for(int i=0; i<N; i++)
        parallel_for(int j=0; j<N; j++) {
            if (i % 1000000 == 0) {
                d_val += d_ar[j];
            }
        }
    return d_val;
}
```

**Figure 1:** A parallel initializer (top) with a naive reverse-mode AD gradient function (bottom) that does not

### Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

William S. Moses<sup>1</sup>, Sri Hari Krishna Narayanan<sup>1</sup>, Ludger Paehler<sup>1</sup>, Valentin Churavy<sup>1</sup>  
MIT CSAIL, Argonne National Laboratory, Technical University of Munich, MIT CSAIL  
wsmoses@mit.edu, snarayan@anl.gov, ludger.paehler@tum.de, vchuravy@mit.edu

Michel Schanen<sup>1</sup>, Jan Hückelheim<sup>1</sup>, Johannes Doerfert<sup>1</sup>, Paul Hovland<sup>1</sup>  
Argonne National Laboratory, Argonne National Laboratory, Argonne National Laboratory, Argonne National Laboratory  
mschanen@anl.gov, jhueckelheim@anl.gov, jdoerfert@anl.gov, hovland@ncsl.anl.gov

**Abstract**—Derivatives are key to numerous science, engineering, and machine learning applications. While existing tools generate derivatives of programs in a single language, modern parallel applications combine a set of frameworks and languages to leverage available performance and function in an evolving hardware landscape. We propose a scheme for differentiating arbitrary DAG-based parallelism that preserves scalability and efficiency, implemented into the LLVM-based Enzyme automatic differentiation framework. By integrating with a full-fledged compiler backend, Enzyme can differentiate numerous parallel frameworks and directly control code generation. Combined with its ability to differentiate any LLVM-based language, this flexibility permits Enzyme to leverage the compiler tool chain for parallel and differentiation-specific optimizations. We differentiate nine distinct versions of the LULESH and multiBEE applications, written in different programming languages (OpenMP, MPI, Julia, CUDA, Fortran, C++, and Rust) on C++ and Fortran, OpenMP, Julia, and multiBEE, distributed, and describe how additional frameworks can be supported by simply marking the parallelism. By enabling support for the underlying programming models within the compiler, we are able to differentiate any parallel framework built on top of them such as RAJA (runtime-agnostic OpenMP and MPI) and MPI/J (Julia bindings for MPI). Moreover, we demonstrate that differentiating low-level parallelism concepts such as shared and break-local memory automatically yields support for higher-level primitives such as reductions or first-private variables. Finally, we showcase how jointly supporting these parallelism models in one tool naturally enables differentiation of hybrid parallel programs, and that deep integration of AD into the compiler enables performance optimizations usually only available in domain-specific/functional programming languages. Overall, our paper makes the following contributions: • An extension to the theory of reverse-mode differentiation of single-static-assignment (SSA) intermediate representations to handle parallel execution of instructions, and thus differentiation of parallel languages and constructs that lower to such a representation. • A demonstration of how implementing this model within the

### Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients

William S. Moses<sup>1</sup>, Valentin Churavy<sup>1</sup>  
MIT CSAIL, MIT CSAIL  
wsmoses@mit.edu, vchuravy@mit.edu

**Abstract**  
Applying differentiable programming techniques and machine learning algorithms to foreign programs requires developers to either rewrite their code in a machine learning framework, or otherwise provide derivatives of the foreign code. This paper presents Enzyme<sup>1</sup>, a high-performance automatic differentiation (AD) compiler plugin for the LLVM compiler framework capable of synthesizing gradients of statically analyzable programs expressed in the LLVM intermediate representation (IR). Enzyme synthesizes gradients for programs written in any language whose compiler targets LLVM IR including C, C++, Fortran, Julia, Rust, Swift, MLIR, etc., thereby providing native AD capabilities in these languages. Unlike traditional source-to-source and operator-overloading tools, Enzyme performs AD on optimized IR. On a machine-learning focused benchmark suite including Microsoft's ADiBench, AD on optimized IR achieves a geometric mean speedup of 4.2 times over AD on IR before optimization allowing Enzyme to achieve state-of-the-art performance. Packaging Enzyme for PyTorch and TensorFlow provides convenient access to gradients of foreign code with state-of-the-art performance, enabling foreign code to be directly incorporated into existing machine learning workflows.

<sup>1</sup>Code and documentation at <https://github.com/wsmoses/Enzyme> and <https://enzyme.mil.edu>.  
34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

# How Does Raising & Tensor Transformations Impact AD?

- Biggest impact in three primary areas:
- Work-Reduction + Fusion
- Checkpointing
- Communication

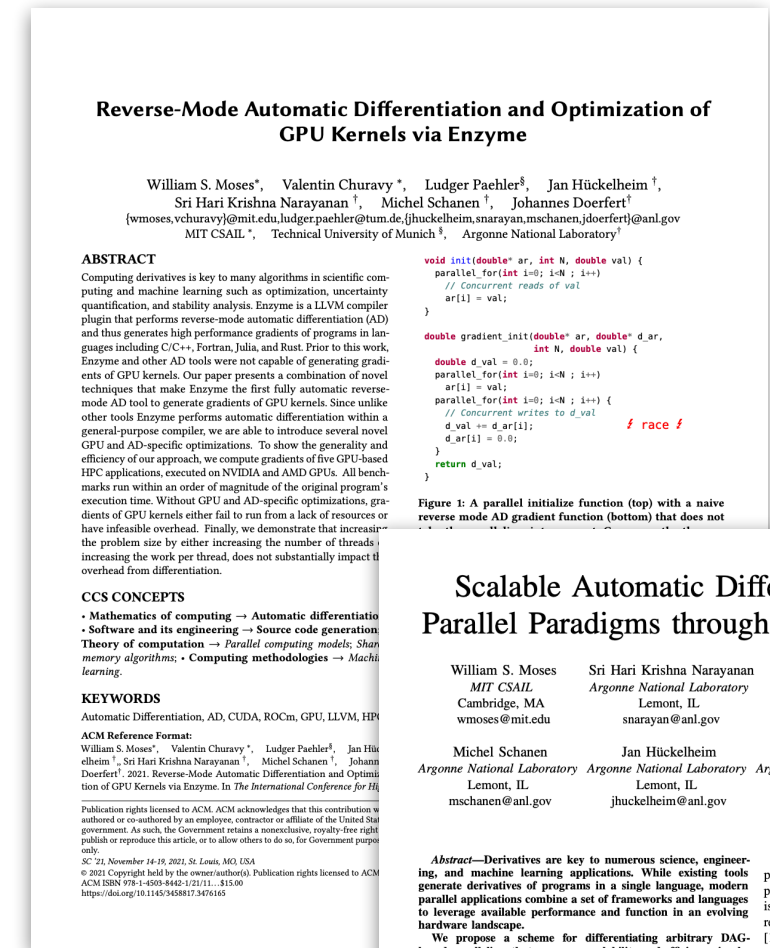
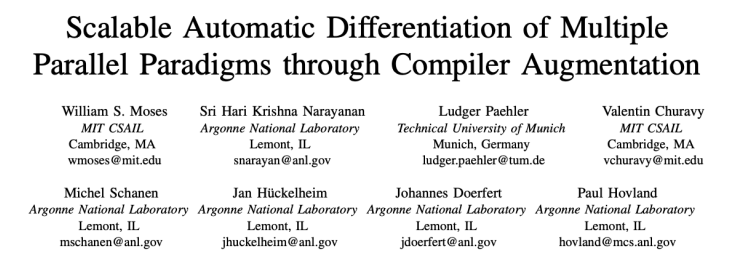


Figure 1: A parallel initializer function (top) with a naive reverse mode AD gradient function (bottom) that does not








item applications in mining [1], gradient [14], and computer AD) is a methodology of mathematical precision, unlike frameworks, has to be compiled and only involves using a Message Passing Interface (MPI) or memory parallelism, or RAA [9].

- A demonstration of how implementing this model within the

<sup>1</sup>Code and documentation at <https://github.com/wsmoses/Enzyme> and <https://enzyme.mil.edu>.

# 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

 <p><b>Martin Lücke</b> University of Edinburgh United Kingdom</p>	 <p><b>Michel Steuer</b> Technische Universität Berlin Germany</p>	 <p><b>Albert Cohen</b> Google DeepMind France</p>
 <p><b>William S. Moses</b> University of Illinois Urbana-Champaign United States</p>	 <p><b>Alex Zinenko</b> Google DeepMind</p>	

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 Steuer 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
```

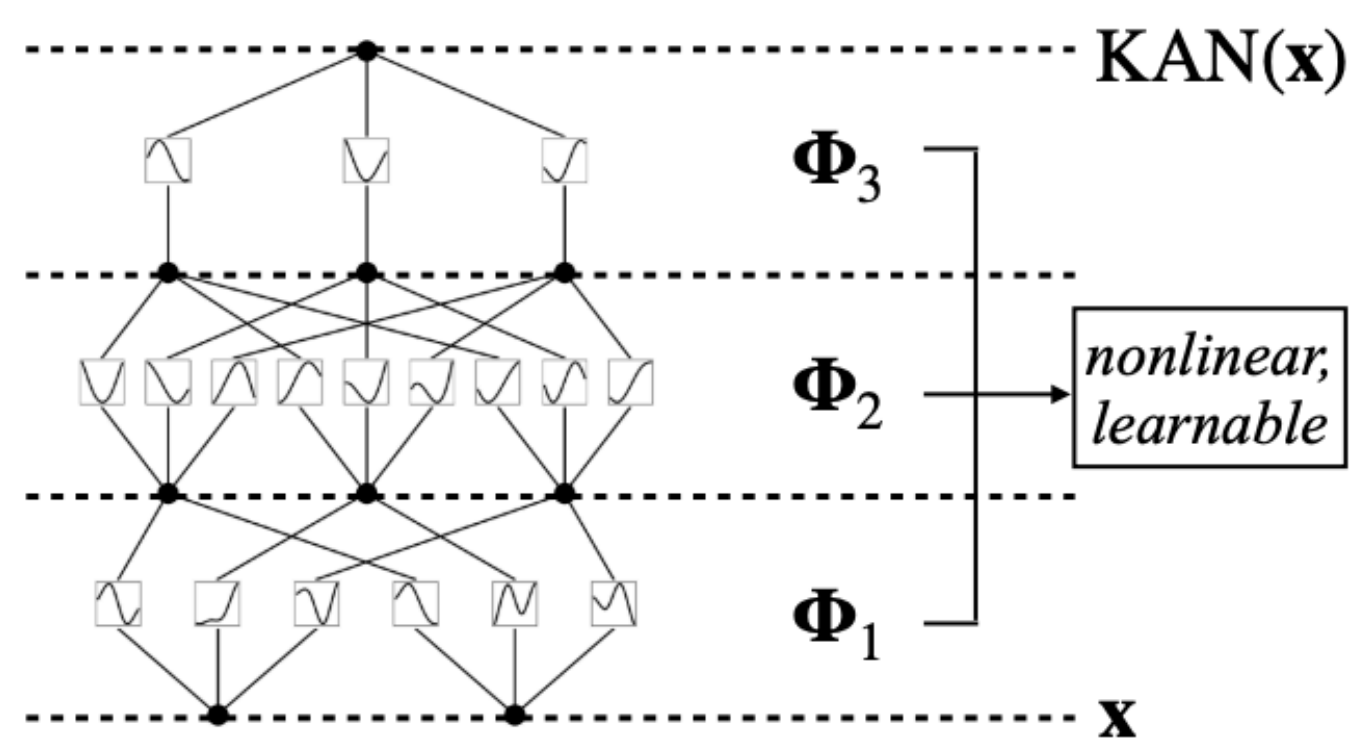
**Unopt: 0.479 samples/sec**

```
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**

# EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)

CUDA KAN network



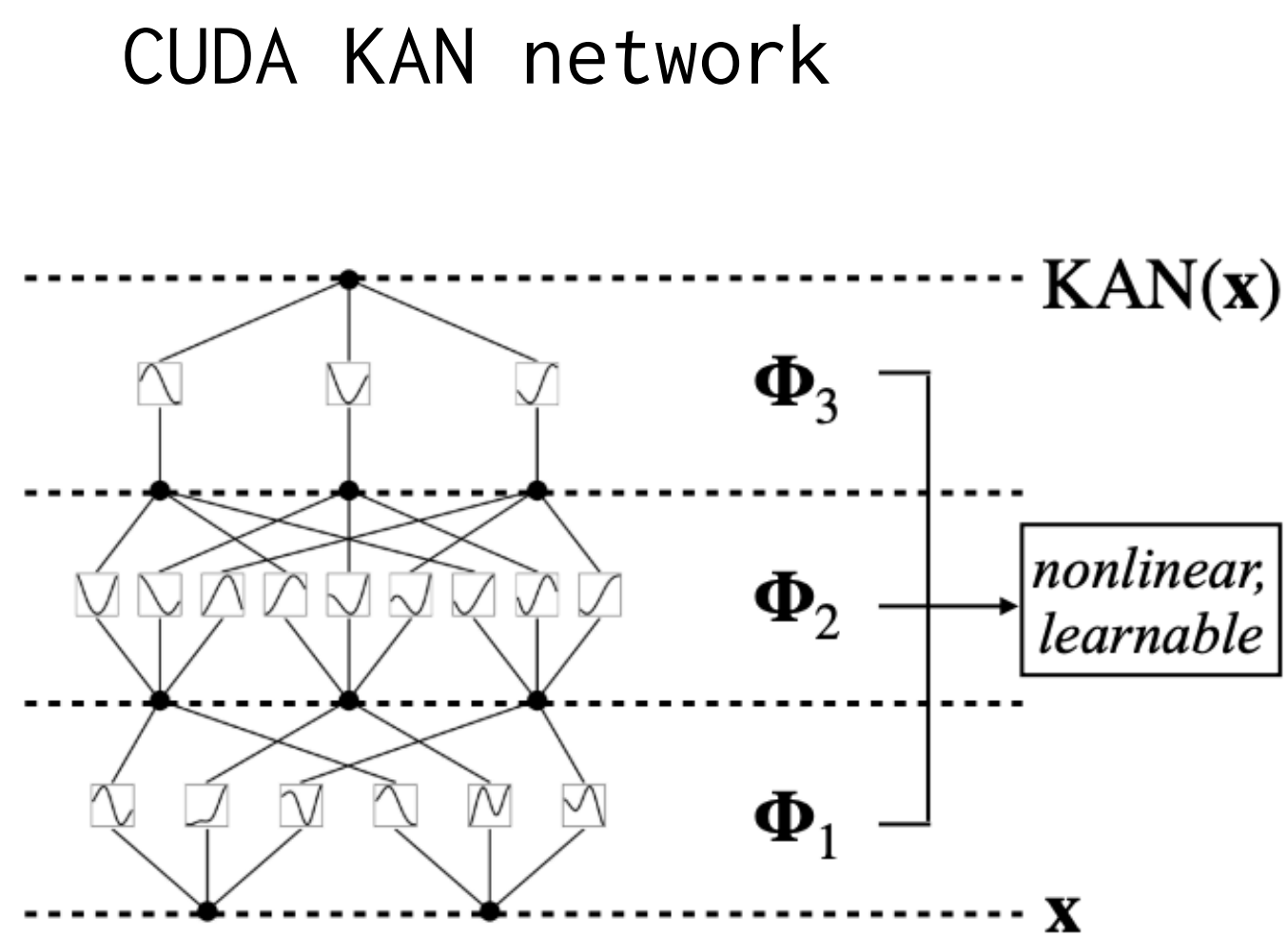
Forward (regular Julia)  
47.586 us ( 248 allocations)  
234.233 us (1022 allocations)  
134.028 us ( 668 allocations)

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)

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)

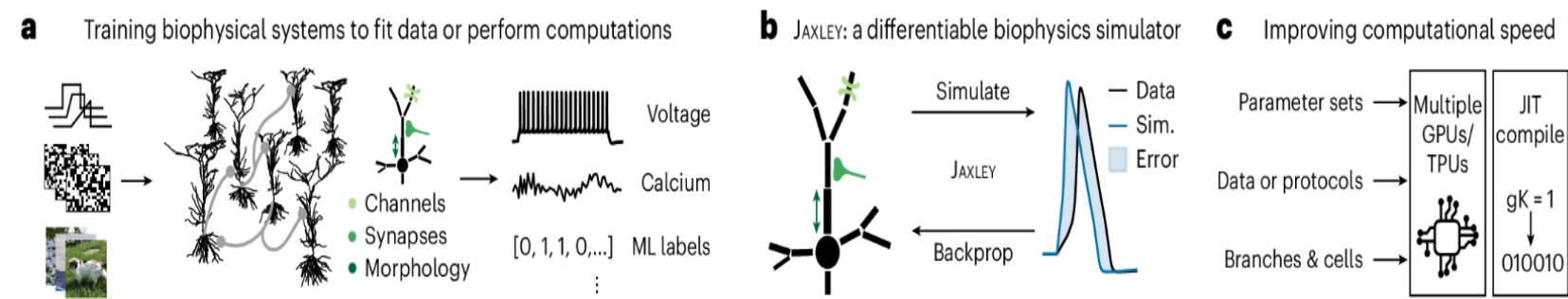
Backwards (EnzymeMLIR + Reactant)  
51.691 us ( 3 allocations)  
104.193 us ( 3 allocations)  
80.020 us ( 3 allocations)

2.14x speedup  
(Primal)

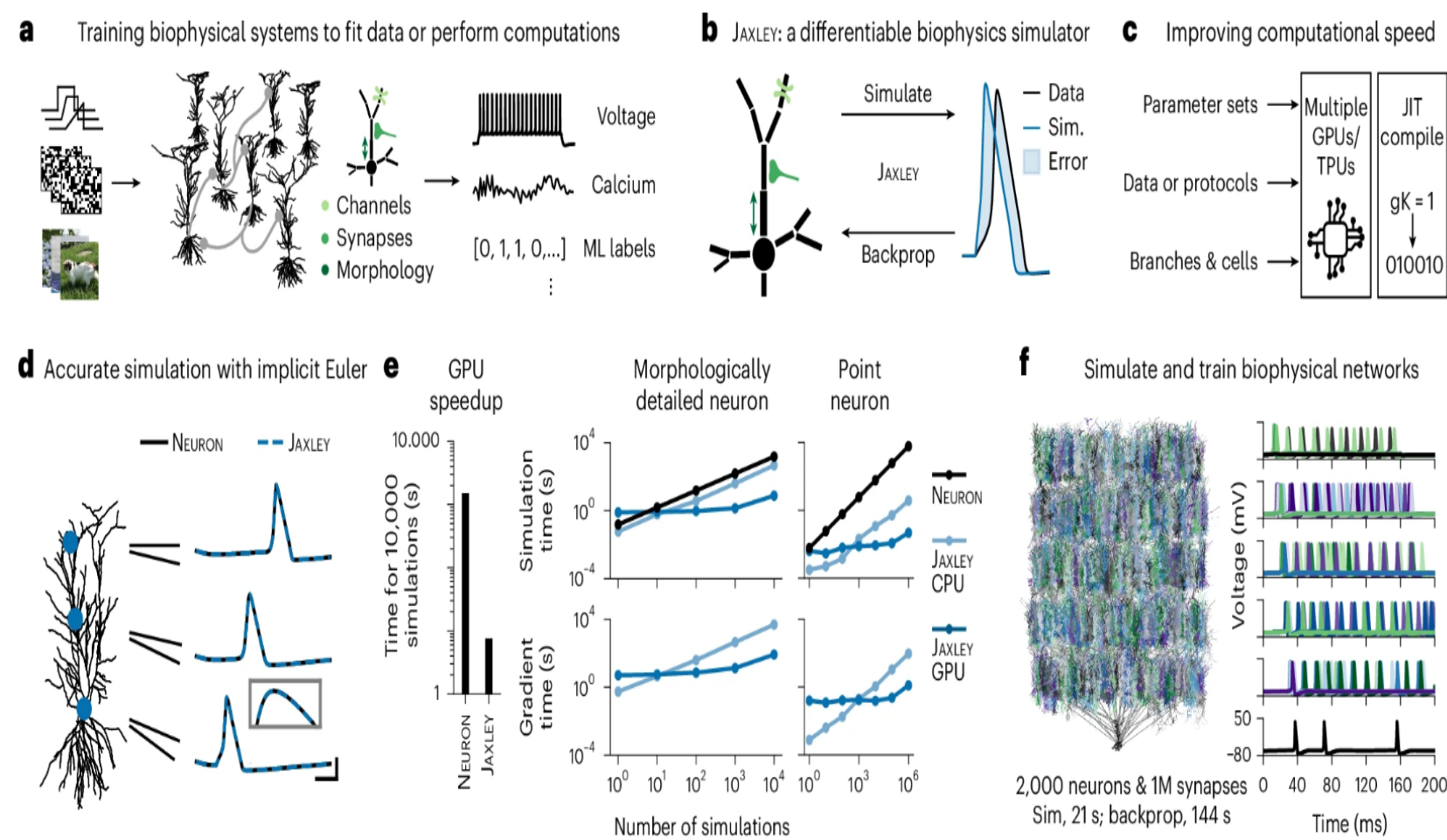
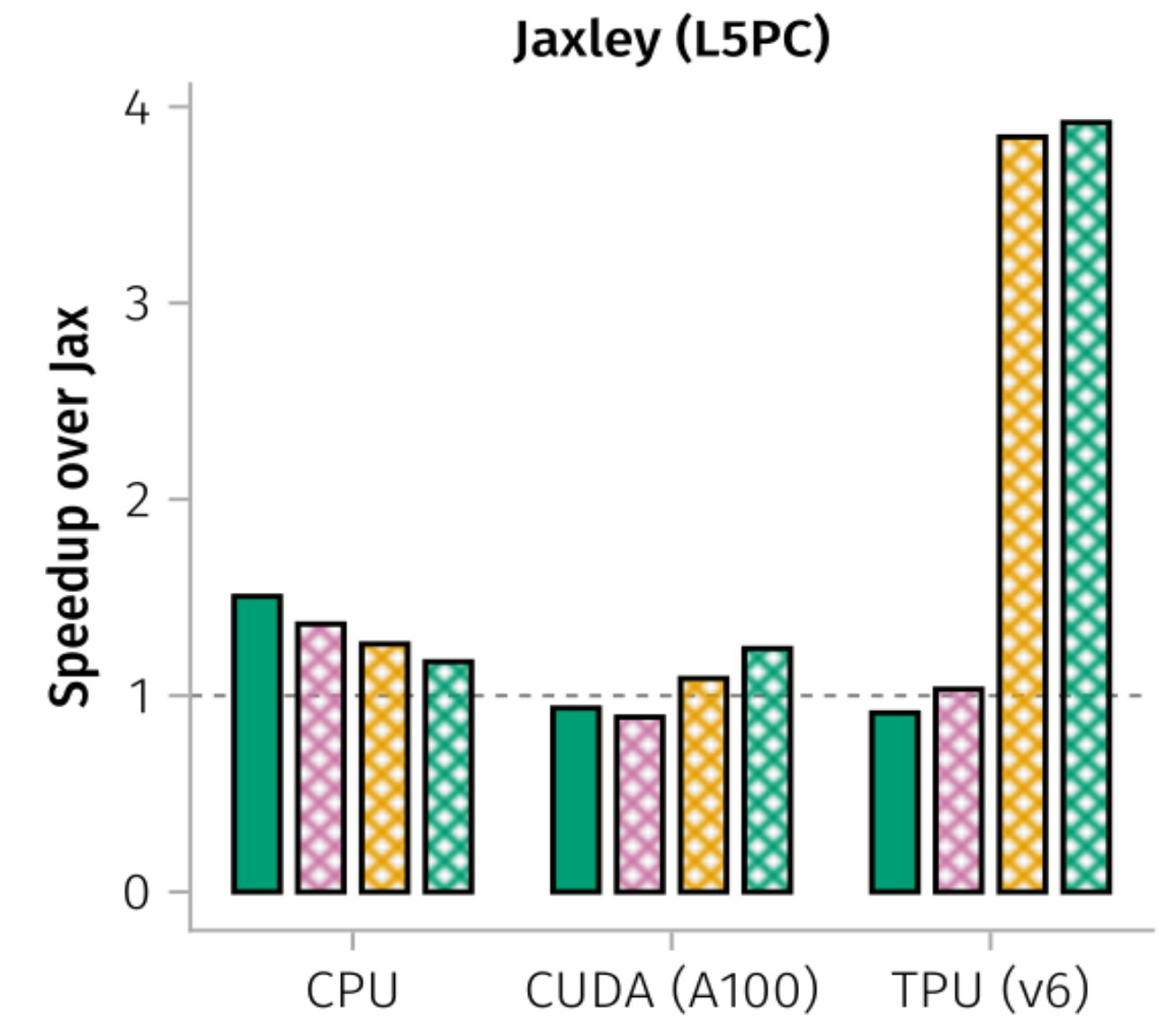
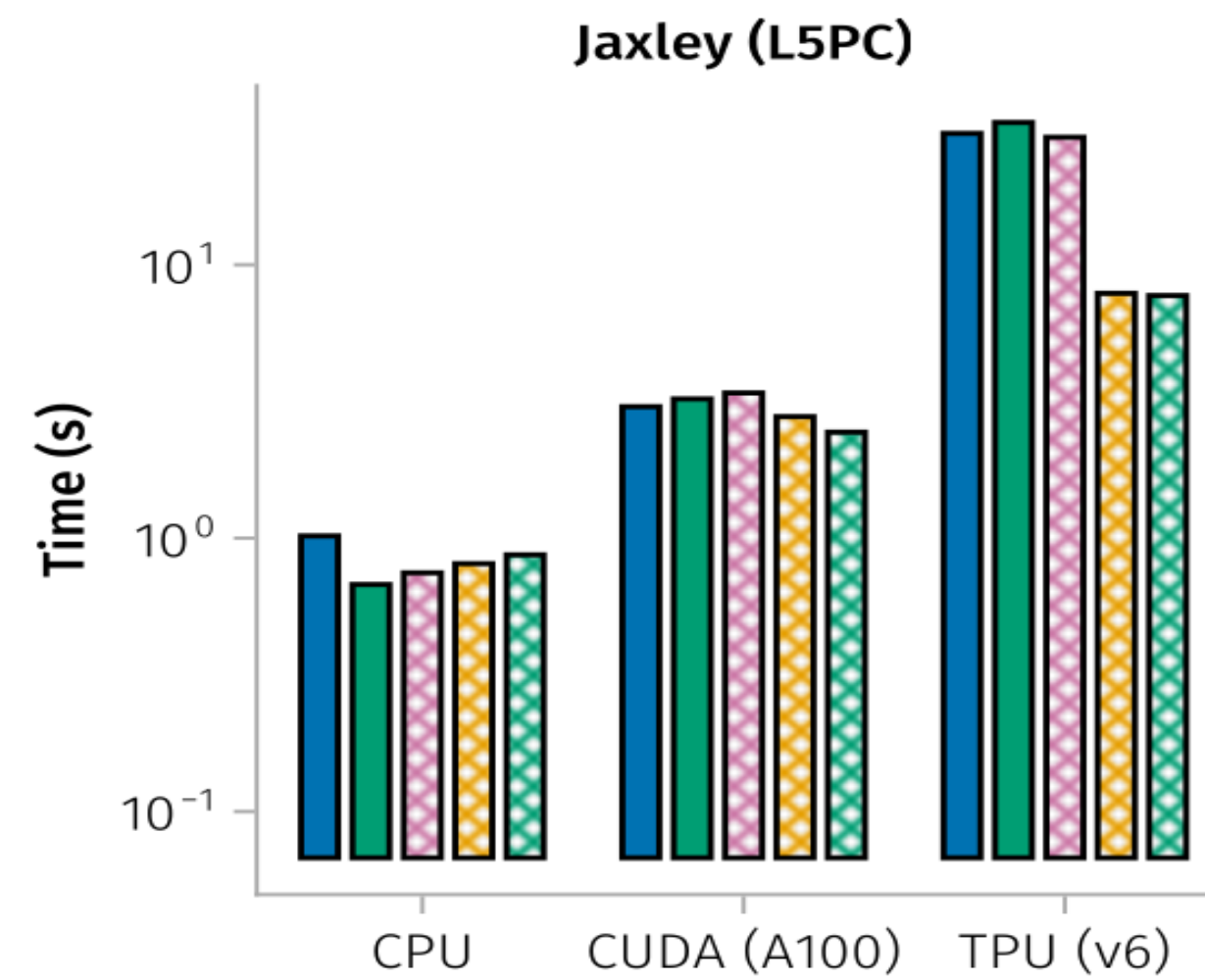
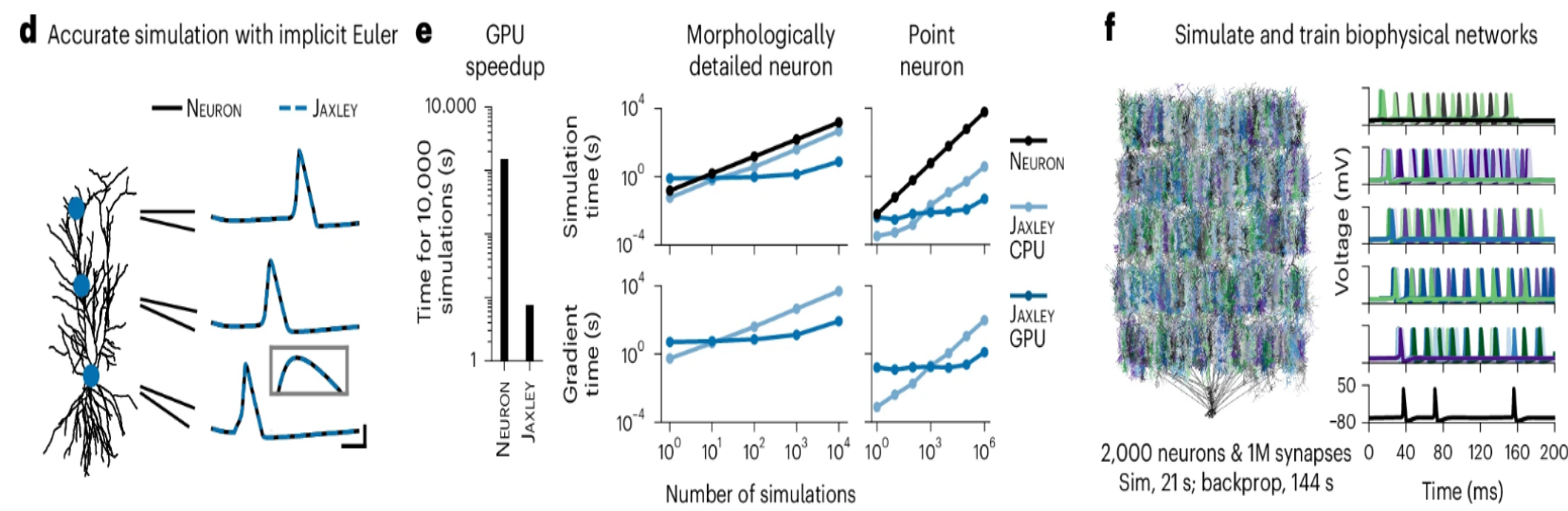


13.57x speedup  
(Derivative)

# Work Reduction Benchmark: Jaxley



1.15x speedup on CPU  
 1.33x speedup on A100  
 3.92x speedup on TPU v6



**scatter\_sum**

**Pass**

- Jax
- HLOOpt
- All optimizations
- Scatter/Gather opts. disabled

**Type**

- No Inlining
- Inlining

# 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](#)



Product

## Claude 3.5 Haiku on AWS Trainium2 and model distillation in Amazon Bedrock

Dec 3, 2024 · 3 min read

## 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

## 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

## OpenAI's Sam Altman is dreaming of Claude 3.5 Haiku on AWS Tr running 100 million GPUs in the future - model distillation in Amazon 100x more than it plans to run by December 2025

Dec 3, 2024 · 3 min read

News

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

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



Comments (0)

## 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.



Comments (2)

# Computing Hardware is No Longer For Everybody

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

By Katie Paul and Krystal Hu

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



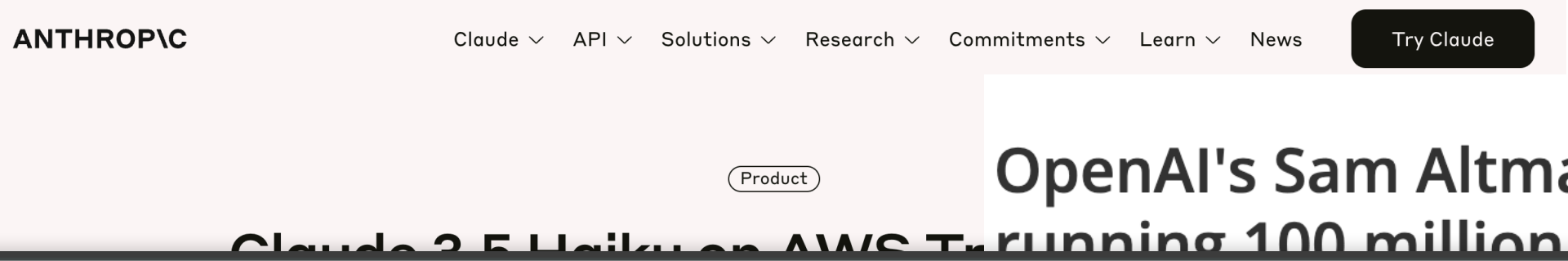
[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium. Photo by Herman/Photo: Purchase Licensing Rights

### Ironwood: The first Google TPU for the age of inference

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

## 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



## OpenAI's Sam Altman is dreaming of running 100 million GPUs in the future - by December

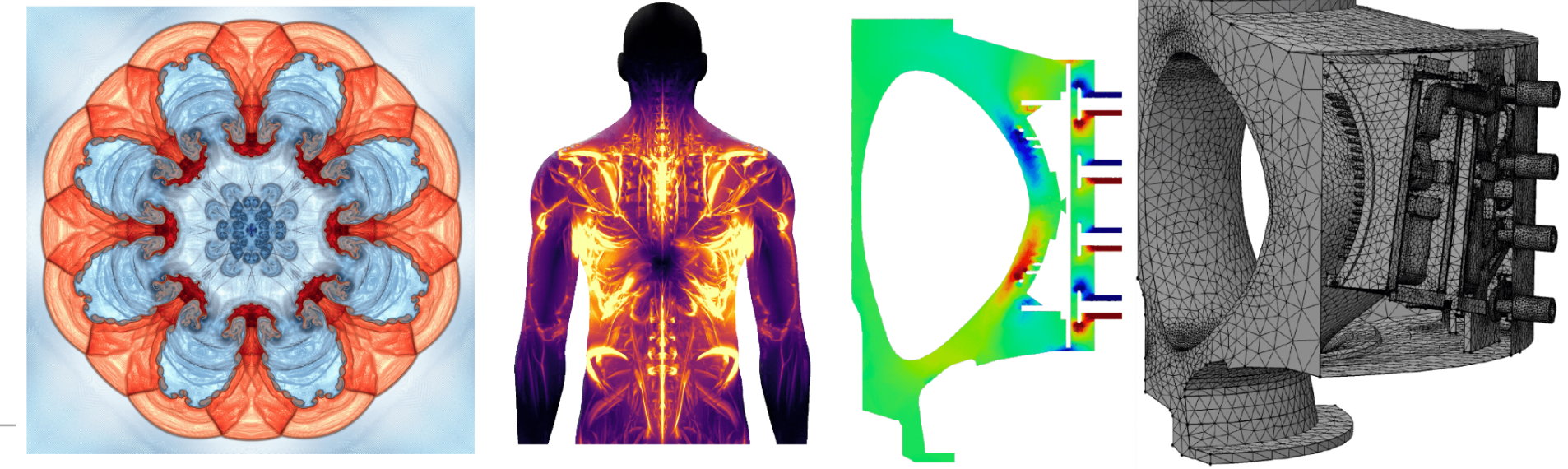
## more NVIDIA GPUs, an impulse all PC gamers can truly understand

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.

 Comments (2)

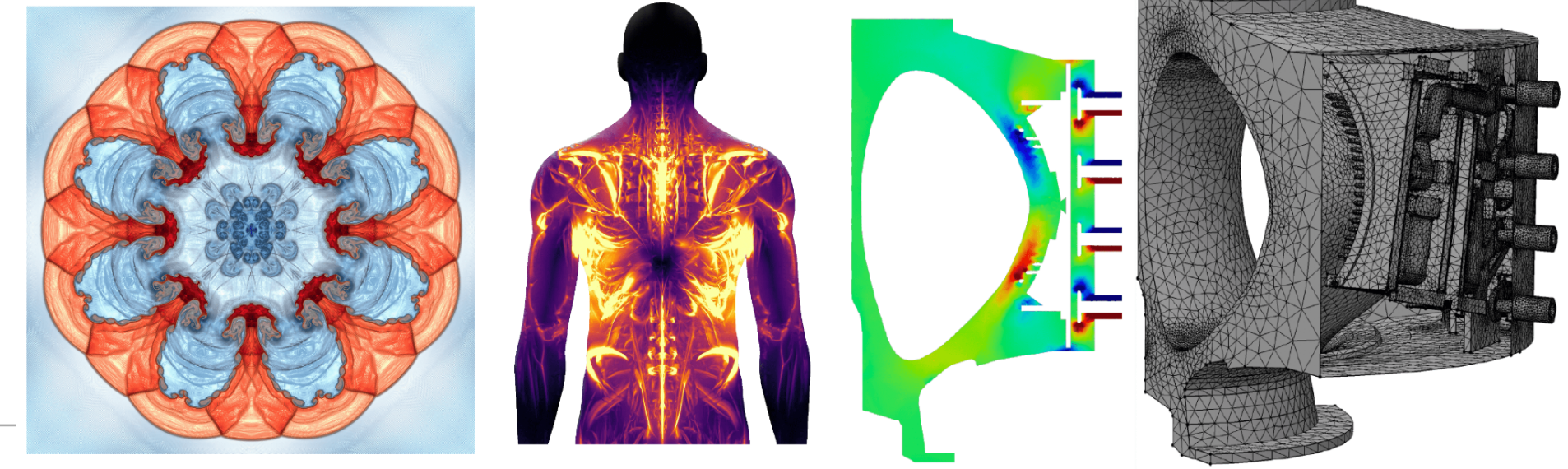
# Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

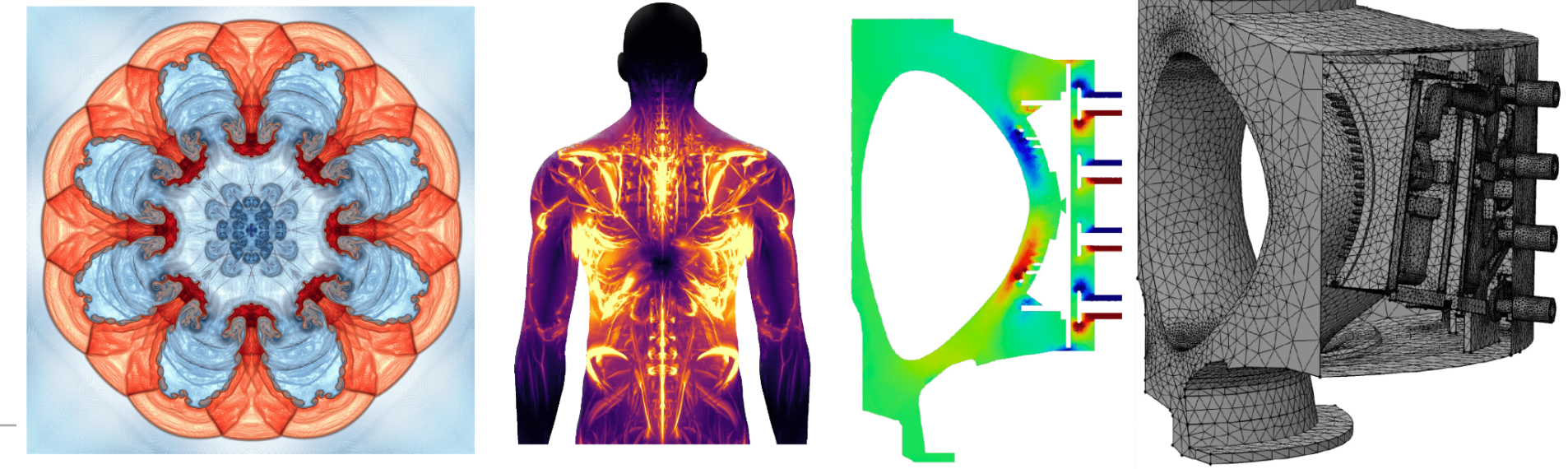
# Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

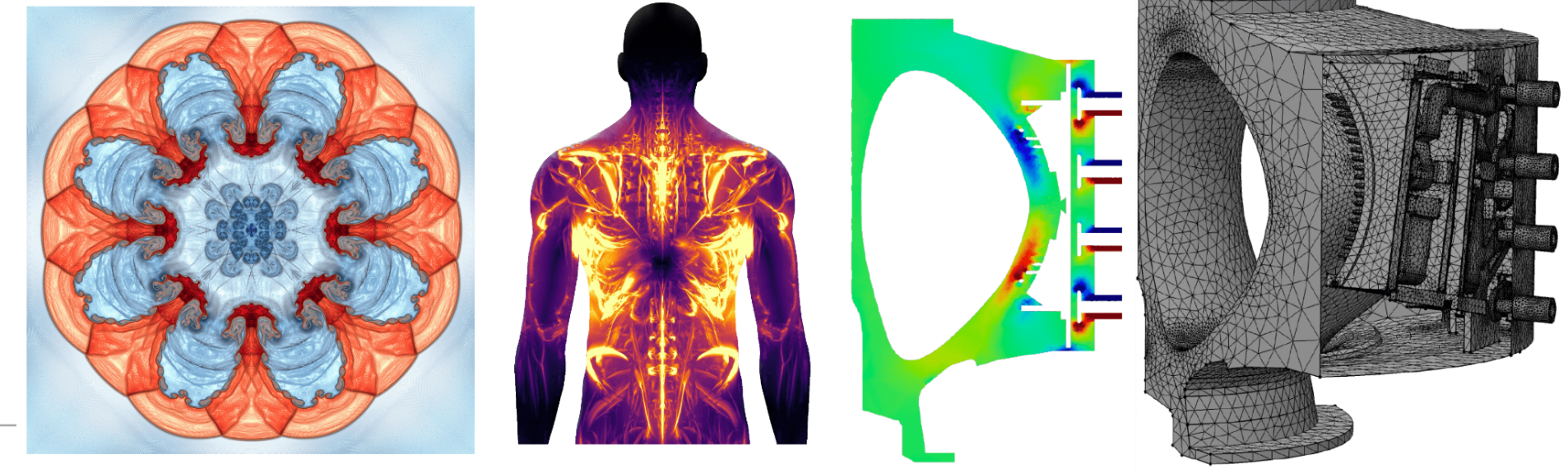
# Lingua Franca of Scientific Computing



- 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;  
    }  
}
```

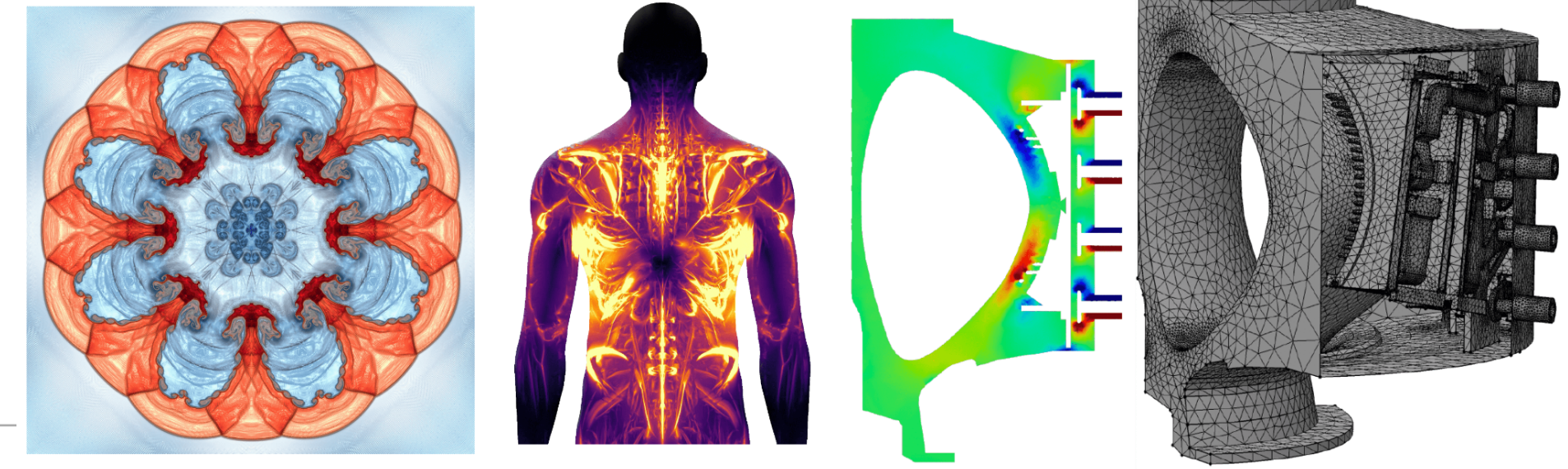
# Lingua Franca of Scientific Computing



- 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);

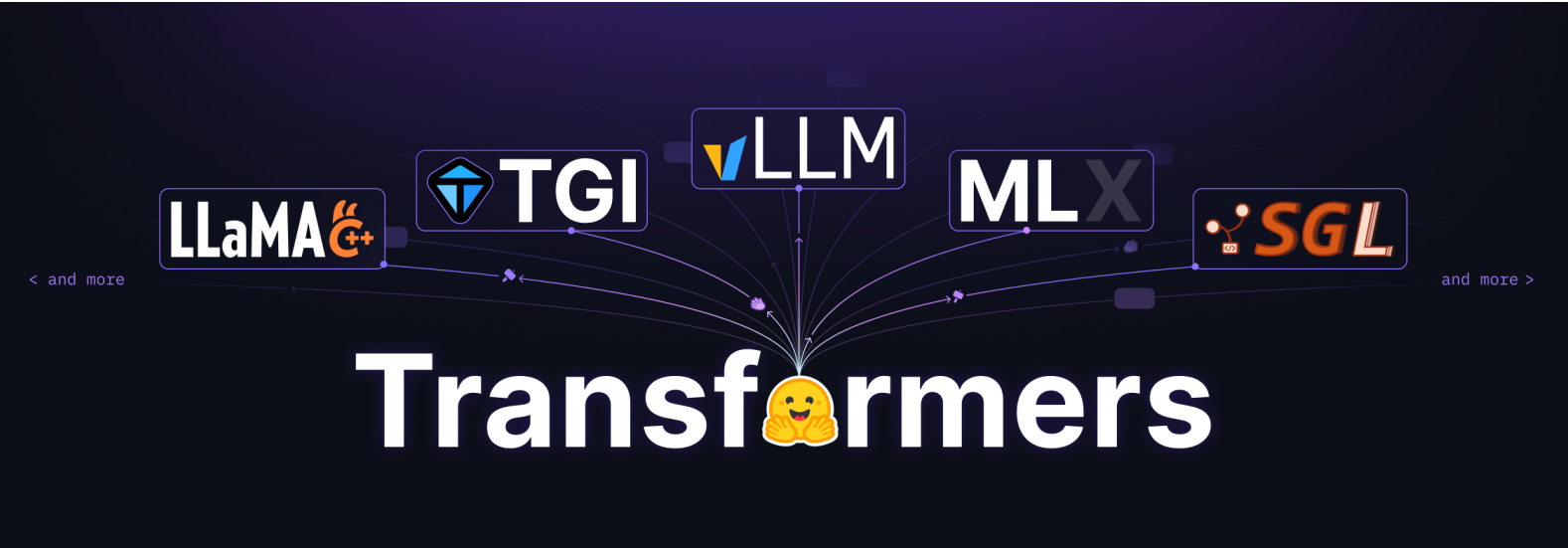
        for (int j=0;j<count;j++)
        {
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
            fx += fx_elem[pos];
            fy += fy_elem[pos];
            fz += fz_elem[pos];
        }

        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?



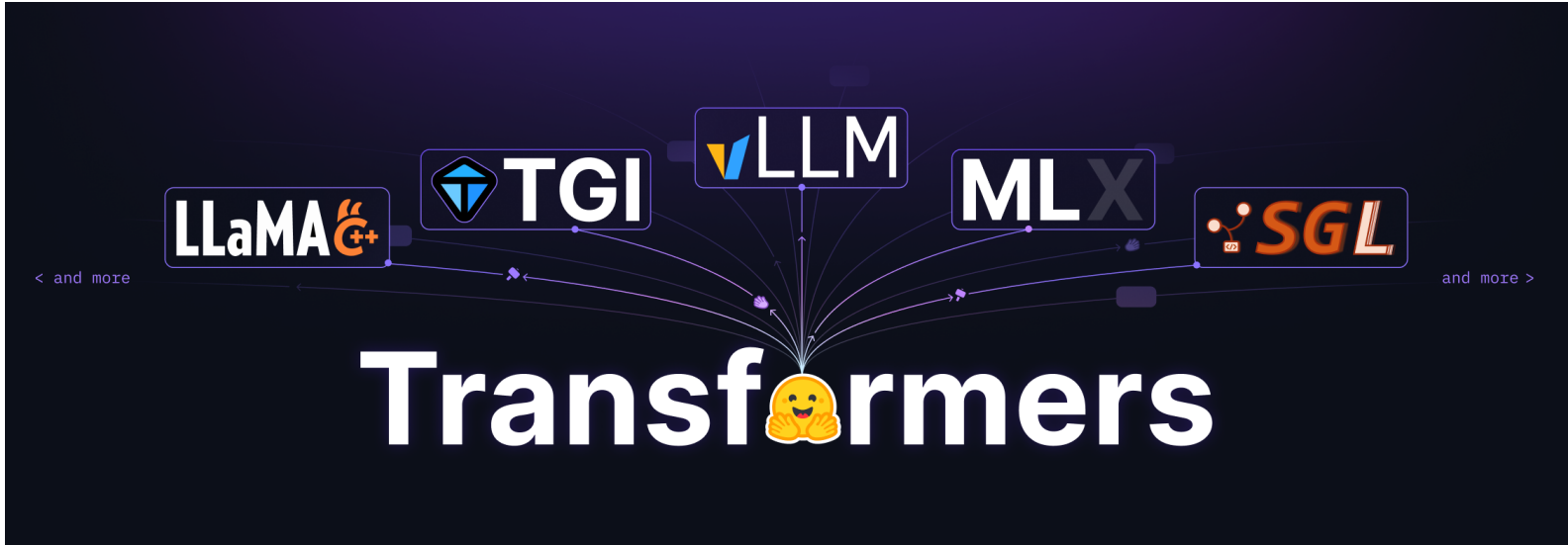
### Stable Diffusion

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](#)

A row of five generated images: a person on a motorcycle in a forest, an astronaut playing a piano, a white unicorn, a dog in a leather jacket, and a bear in a space suit.


# 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](#)




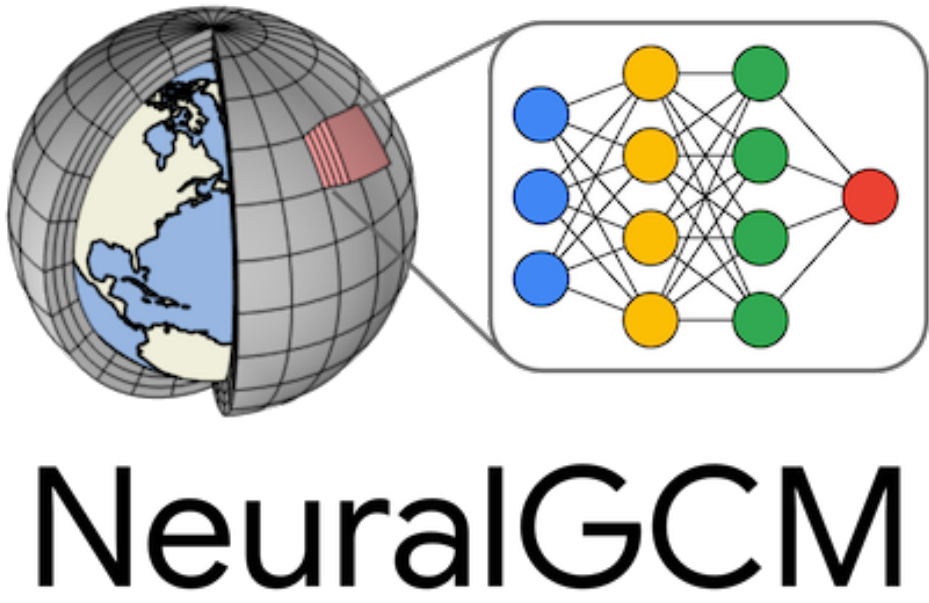
### JAX, M.D.

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



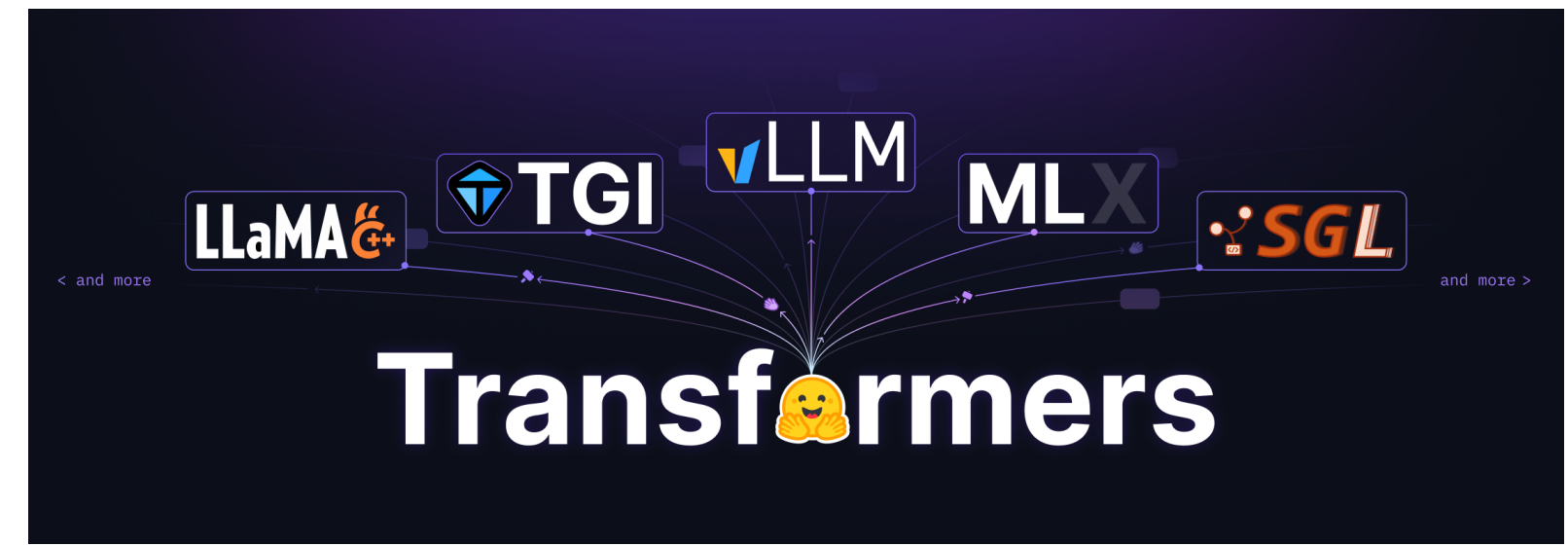
### jaxspec

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.


# 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](#)



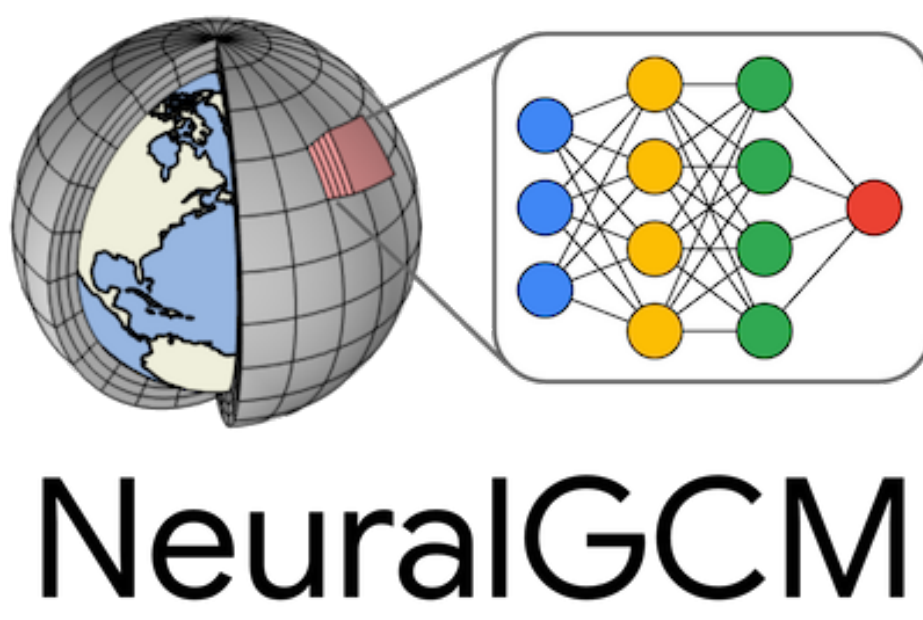
### JAX, M.D.

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



The image shows the 'jaxspec' logo with a cat icon and a badge with the following text: 'PYPI v0.3.0 PYTHON &gt;=3.10,&lt;3.13 DOCS PASSING COVERAGE 94% SLACK'. Below the badge is a warning icon and text: '⚠️ jaxspec is still in early release: expect bugs, breaking API changes, undocumented features and lack of functionalities'. At the bottom, a paragraph describes jaxspec as an X-ray spectral fitting library built in pure Python.

## 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.



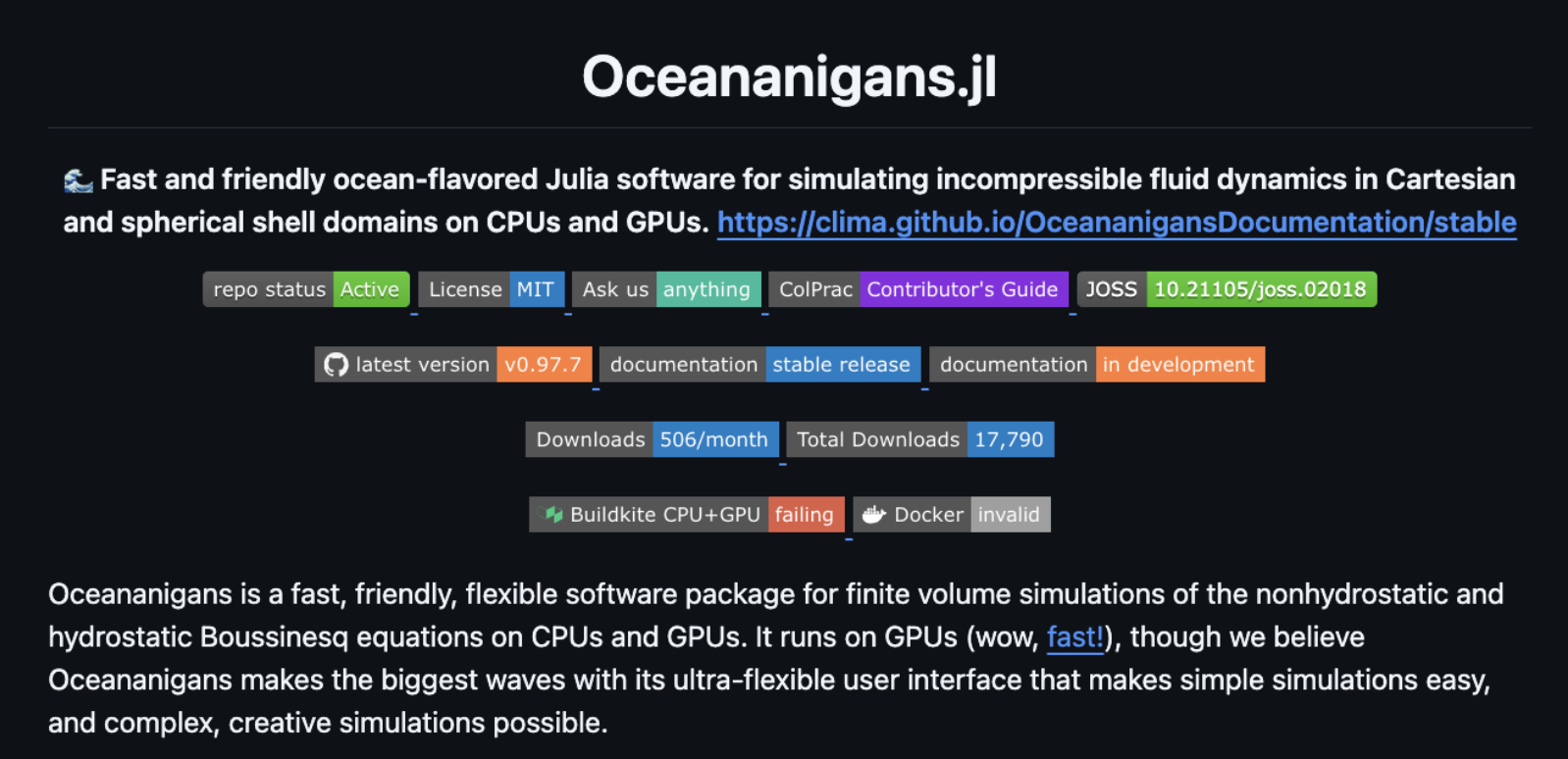
**First and only** open-source scientific software stack developed for scalability and available across all HPC platforms, including cloud computing.

# 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**

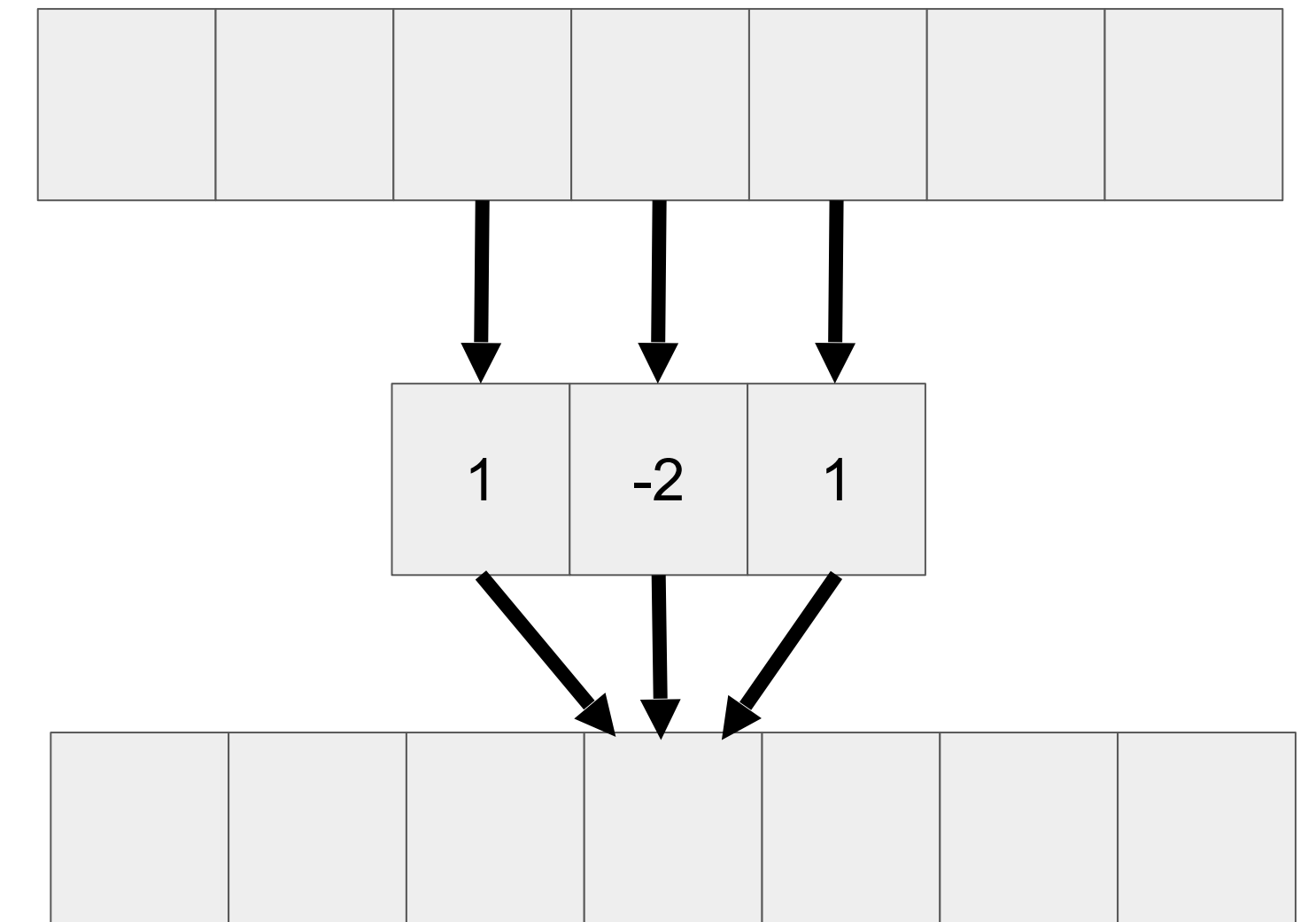
Oceananigans is a fast, friendly, flexible software package for finite volume simulations of the nonhydrostatic and hydrostatic Boussinesq equations on CPUs and GPUs. It runs on GPUs (wow, **fast!**), though we believe Oceananigans makes the biggest waves with its ultra-flexible user interface that makes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

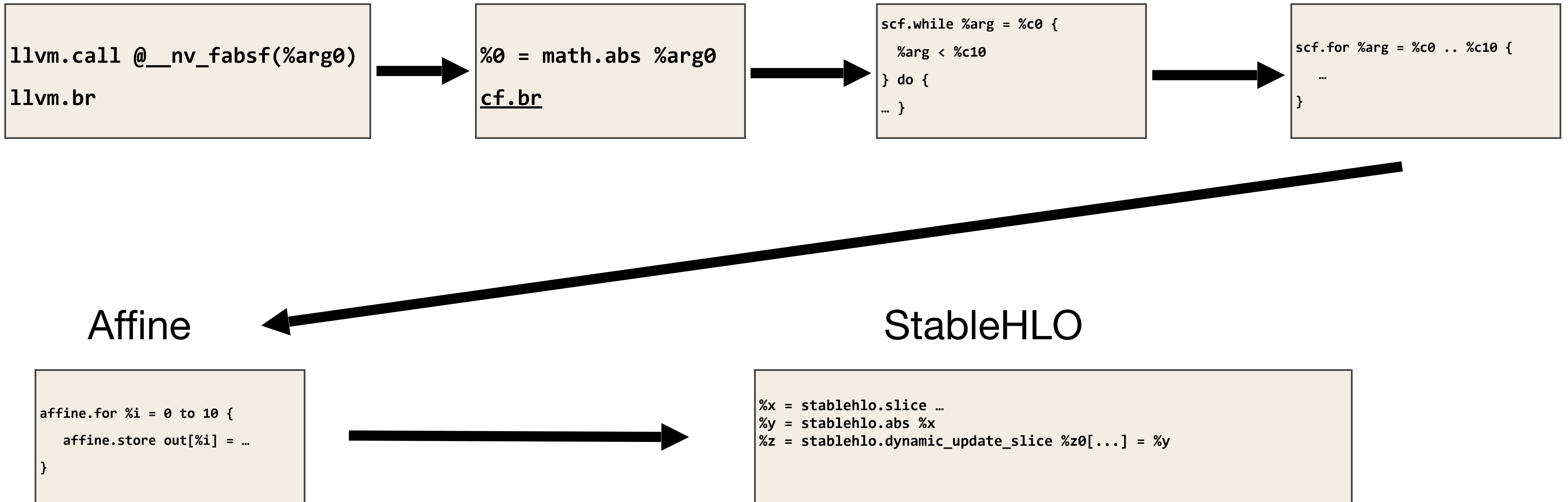
```
scf.for %arg = %c0 .. %c10 {
  ...
}
```

Affine

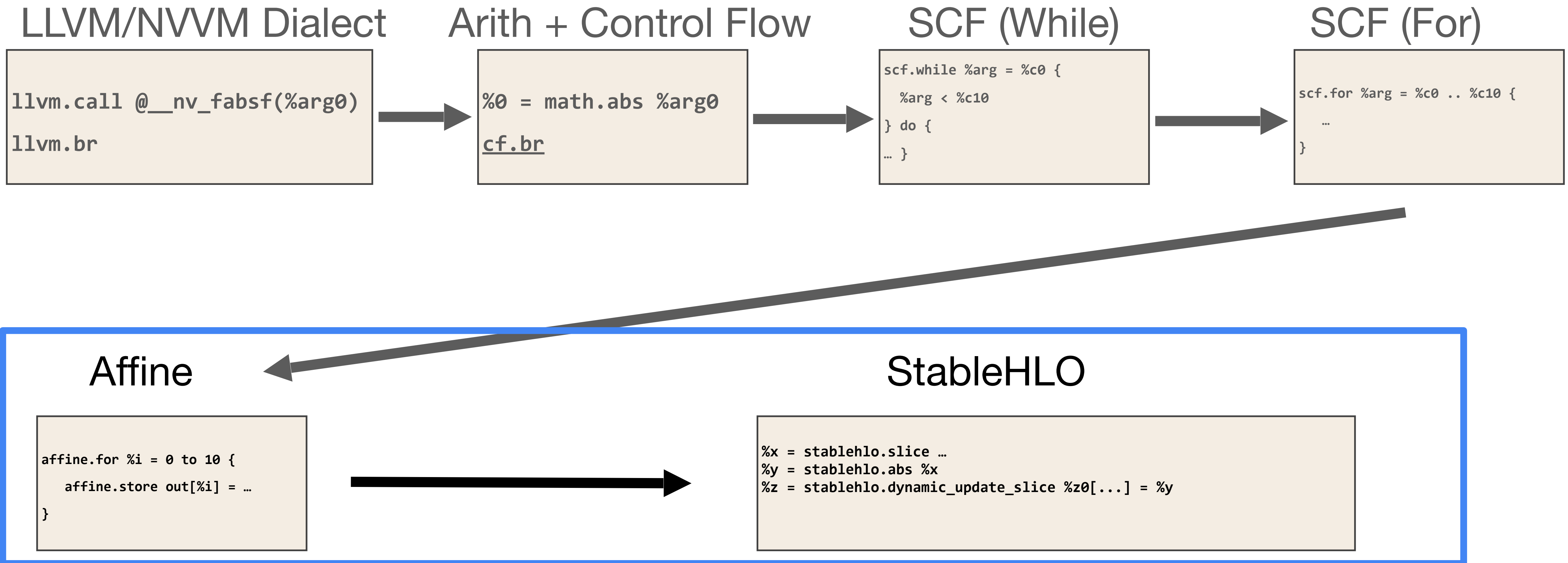
```
affine.for %i = 0 to 10 {
  affine.store out[%i] = ...
}
```

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 will lost the convolution!
- Perform tensor-level optimizations on stablehlo to recover and optimize higher-level structures

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

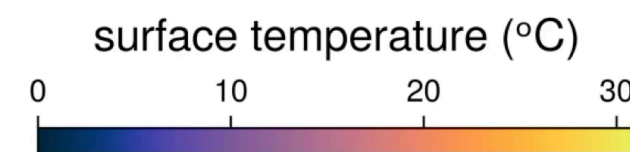
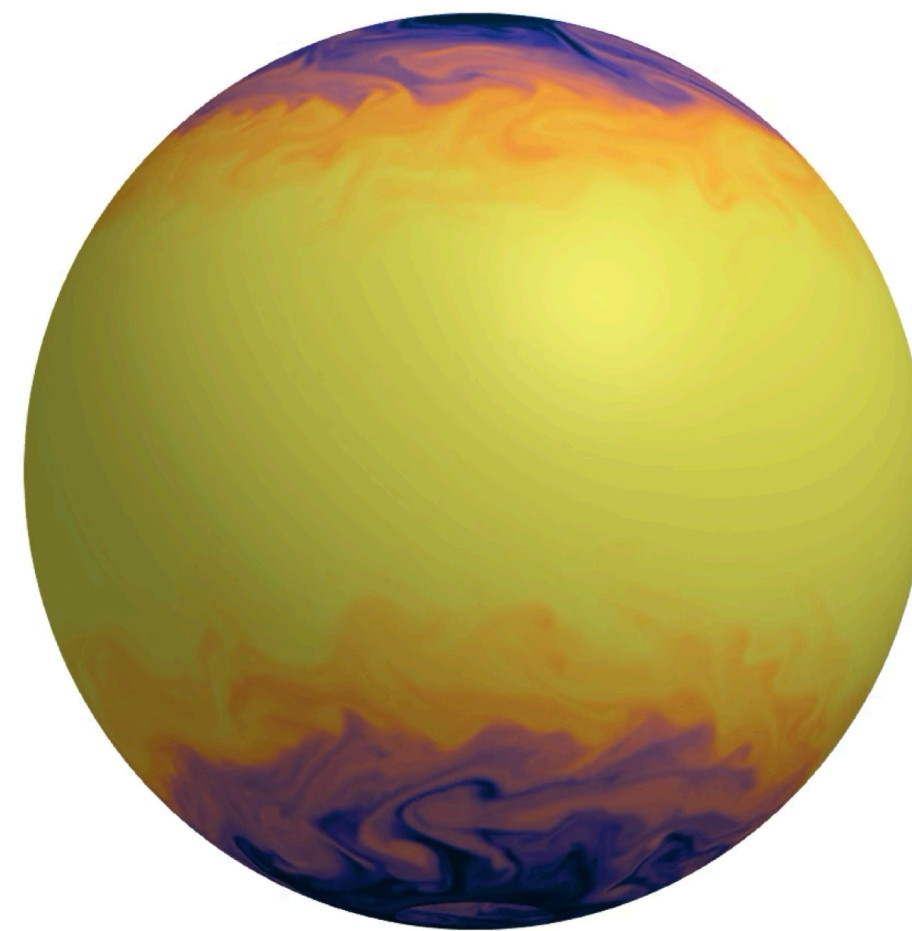
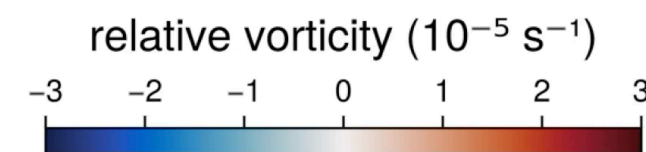
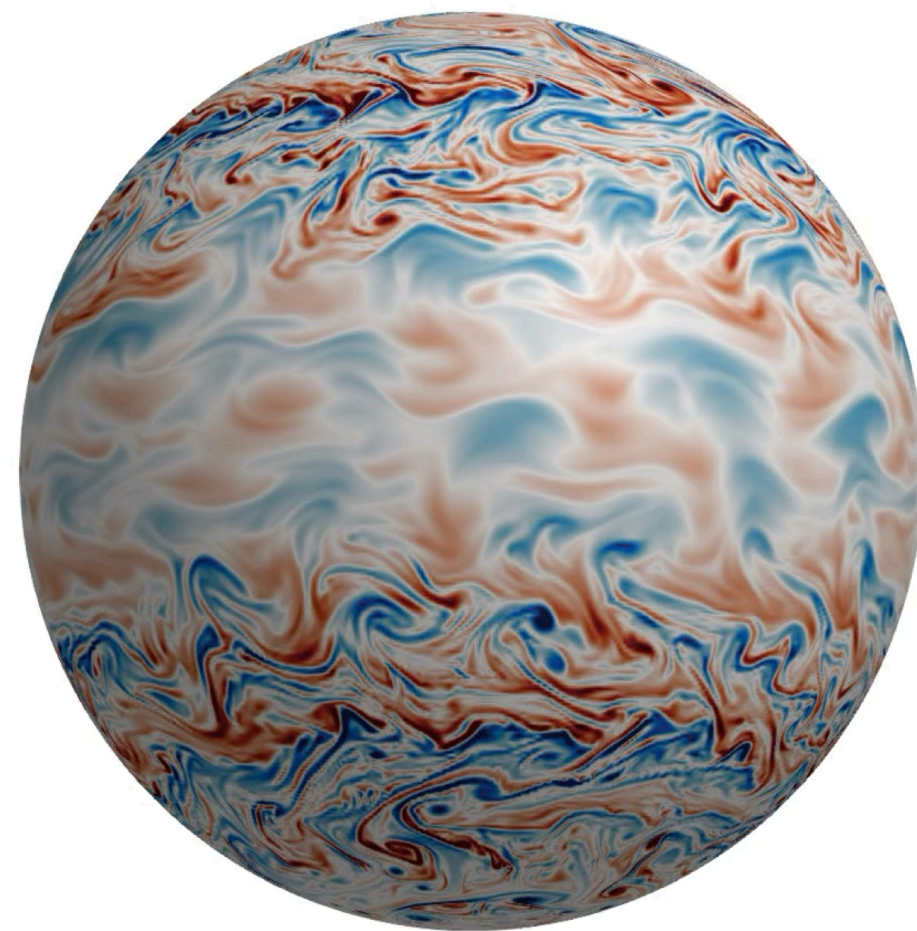
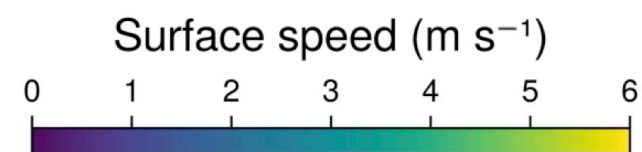
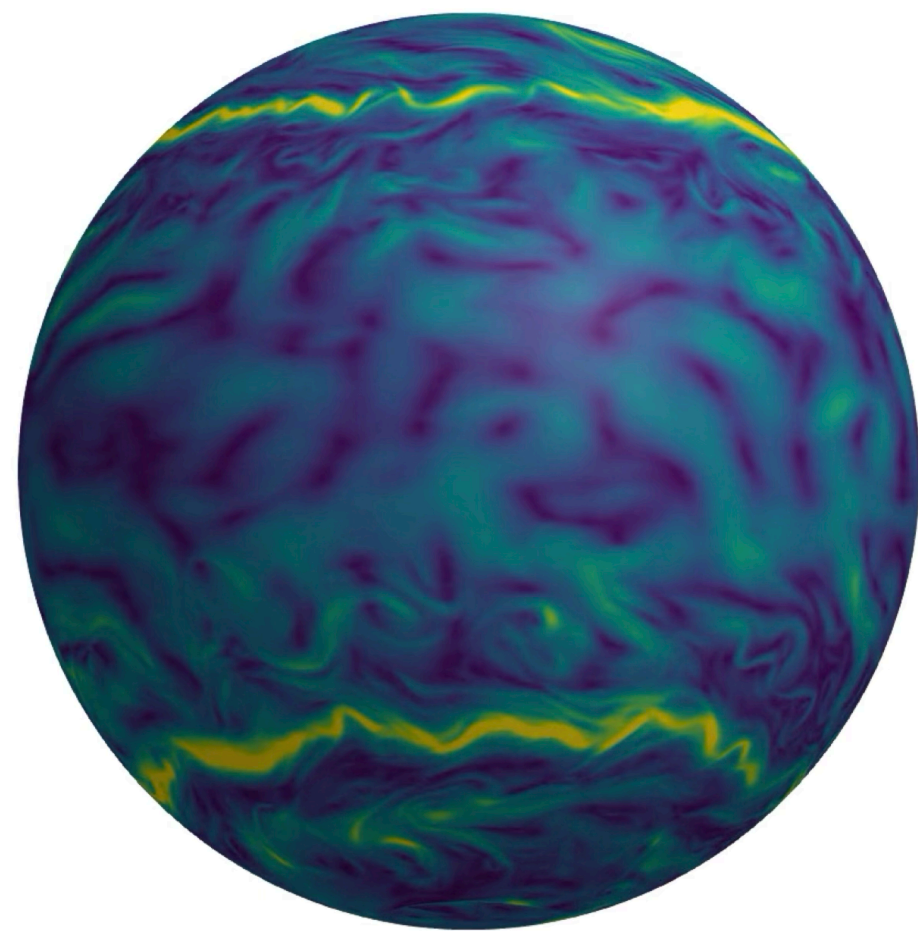
```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>  
%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```

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



# 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

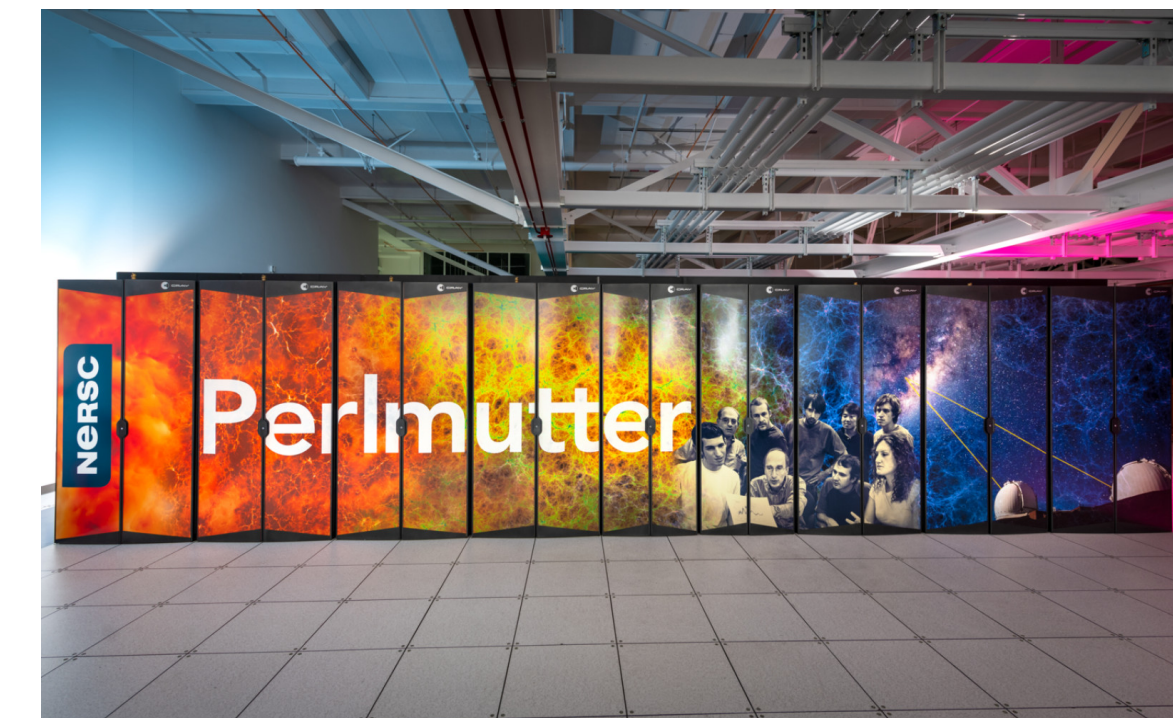
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mul
```

Optimization

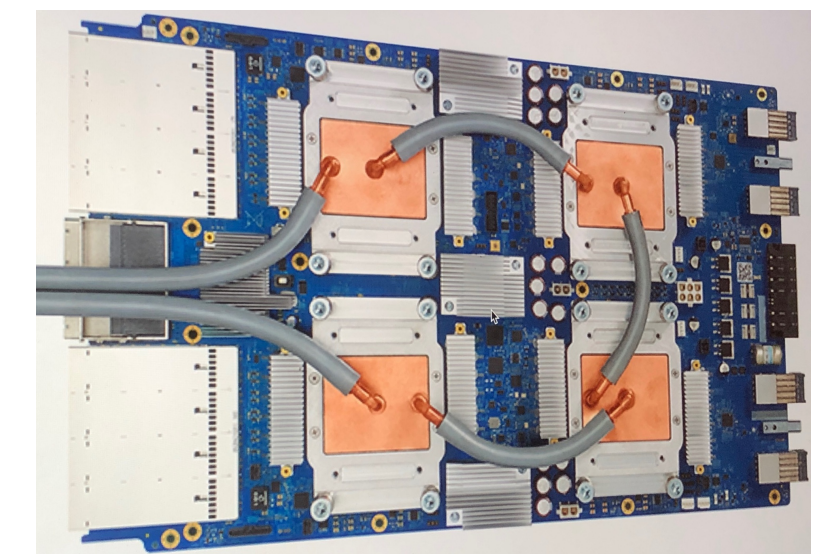
```
res = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```

# Performance Results

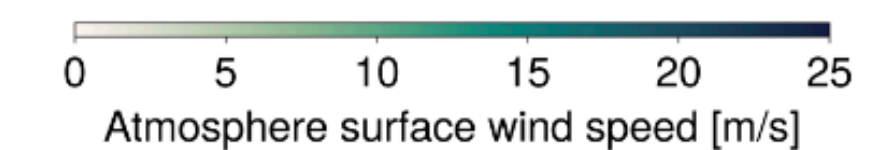
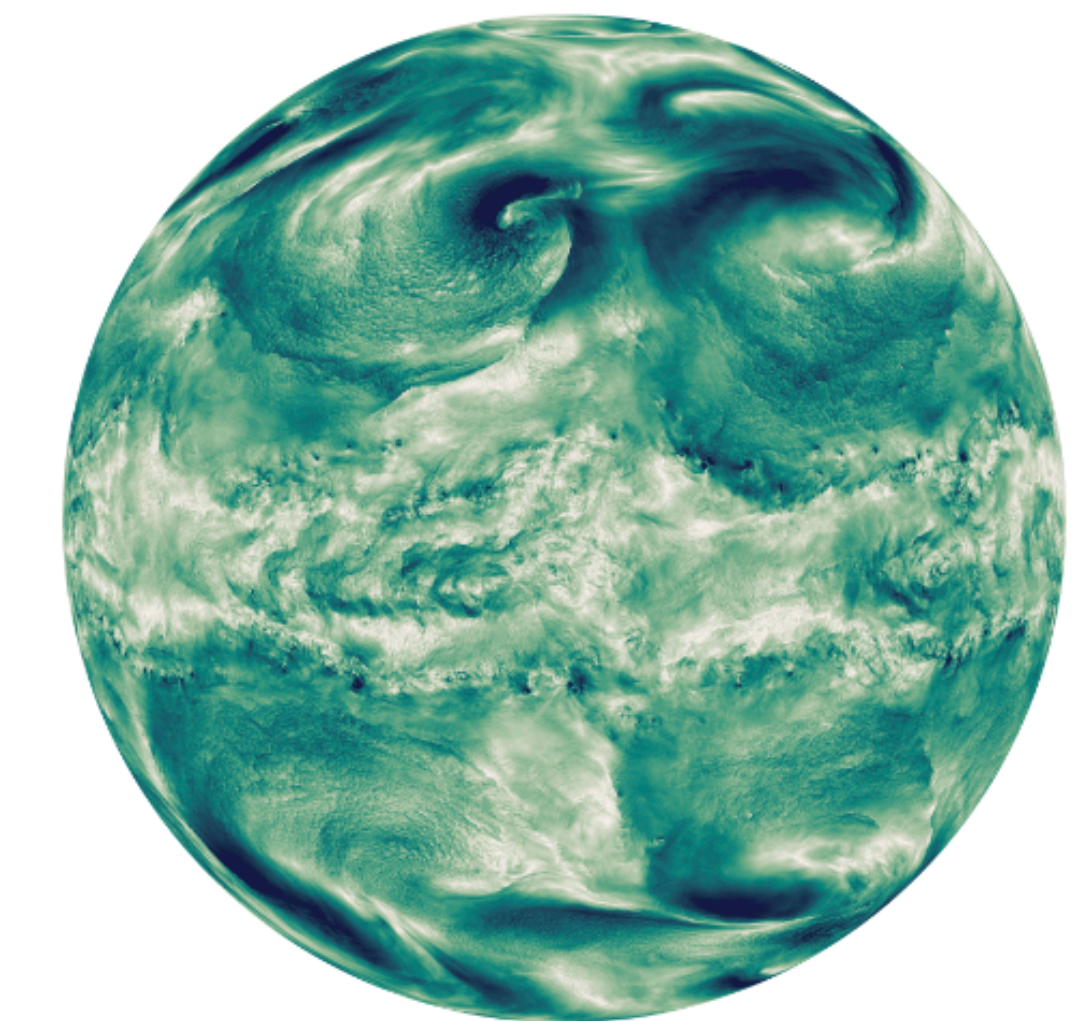
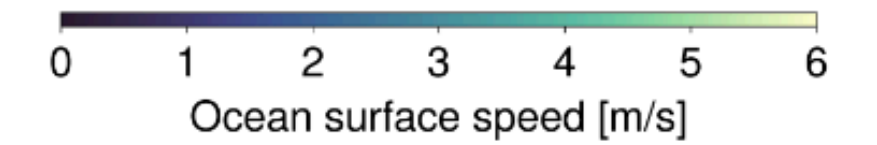
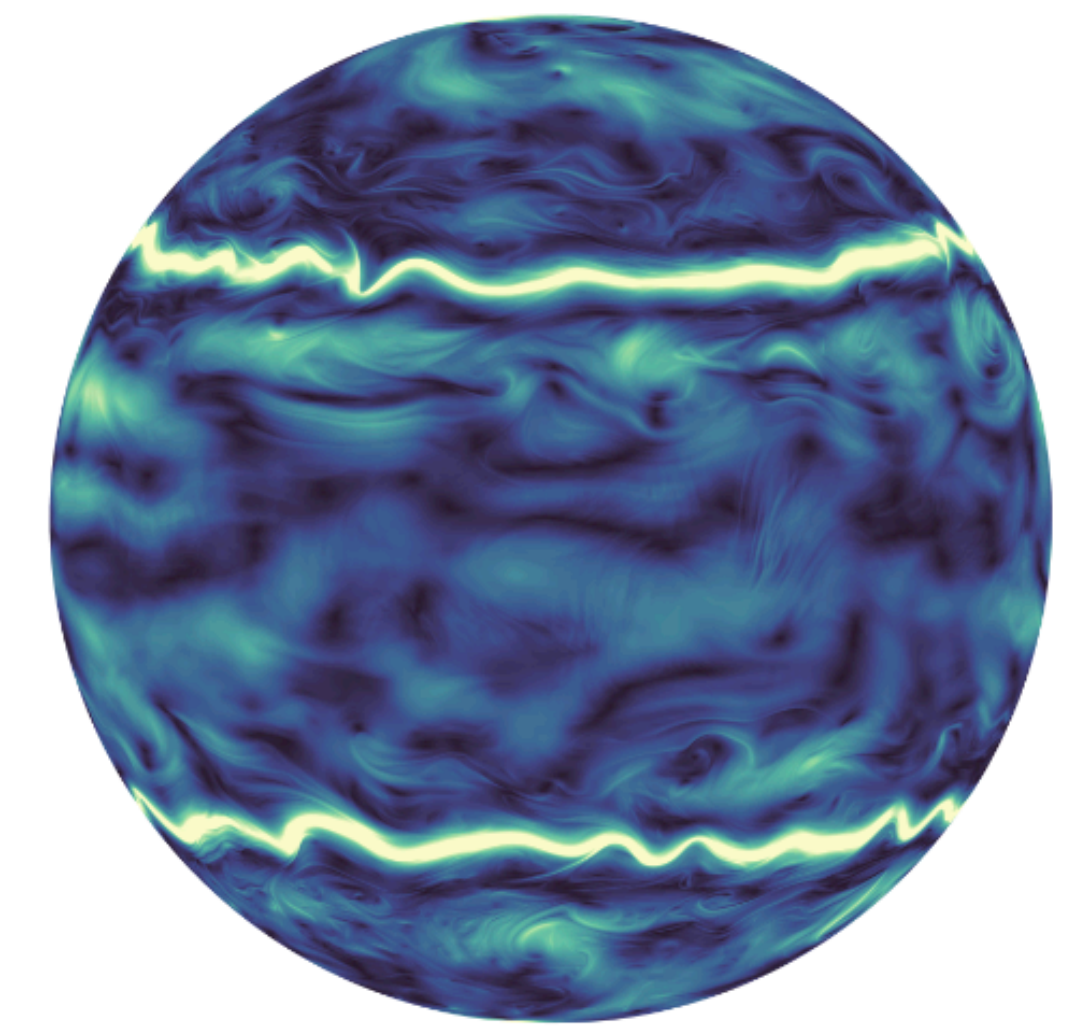
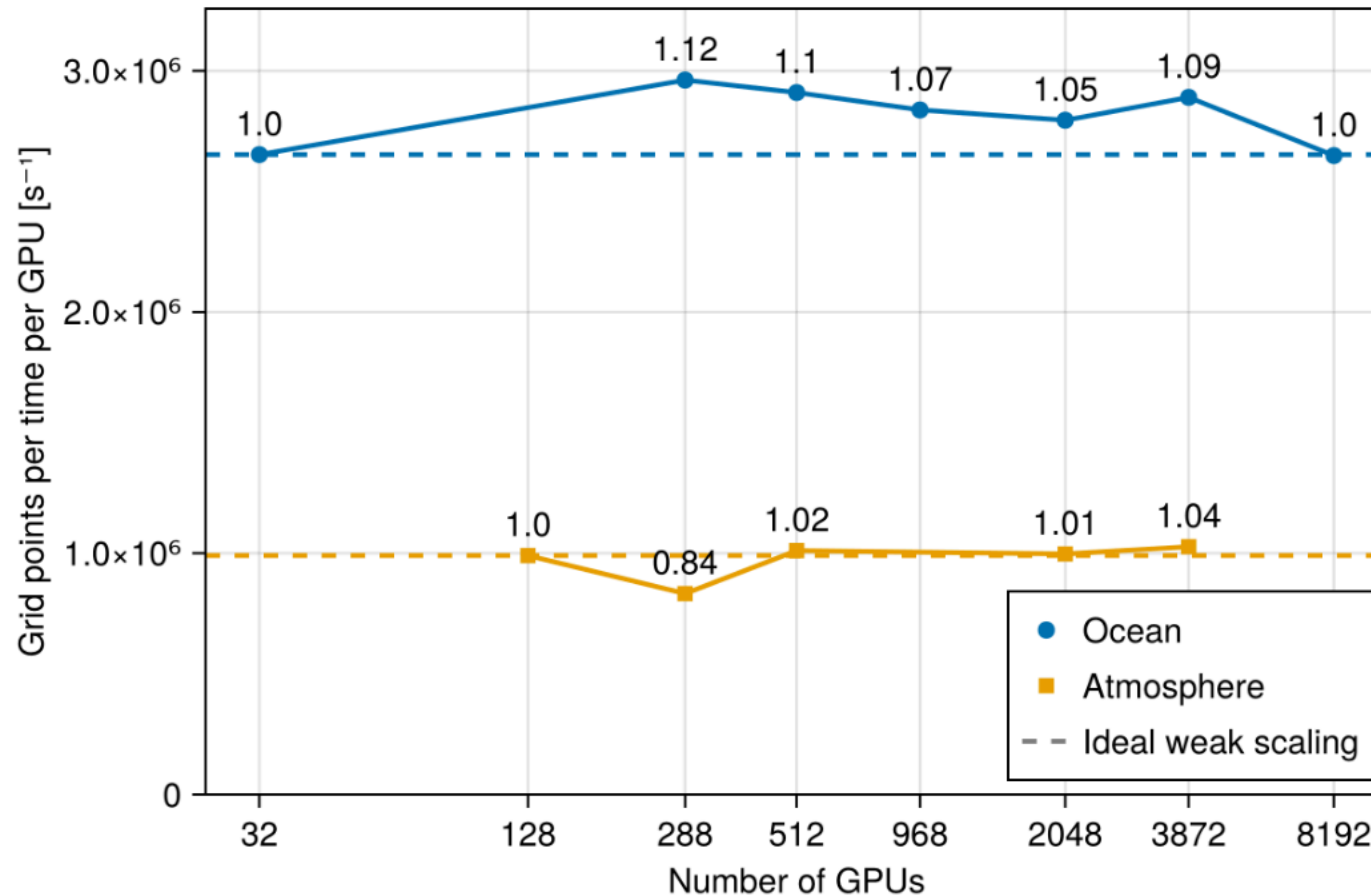
- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
  - Alps (2688 nodes x 4 NVIDIA GH200 GPUs)
  - Perlmutter (1536 nodes x 4 NVIDIA A100 GPUs)
  - 8,192 Google TPUs v7 (4614 F8 TFLOPS each)
- Good Single-Node Perf (CPU)
  - Vanilla Model: 272.0seconds
  - Tensor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



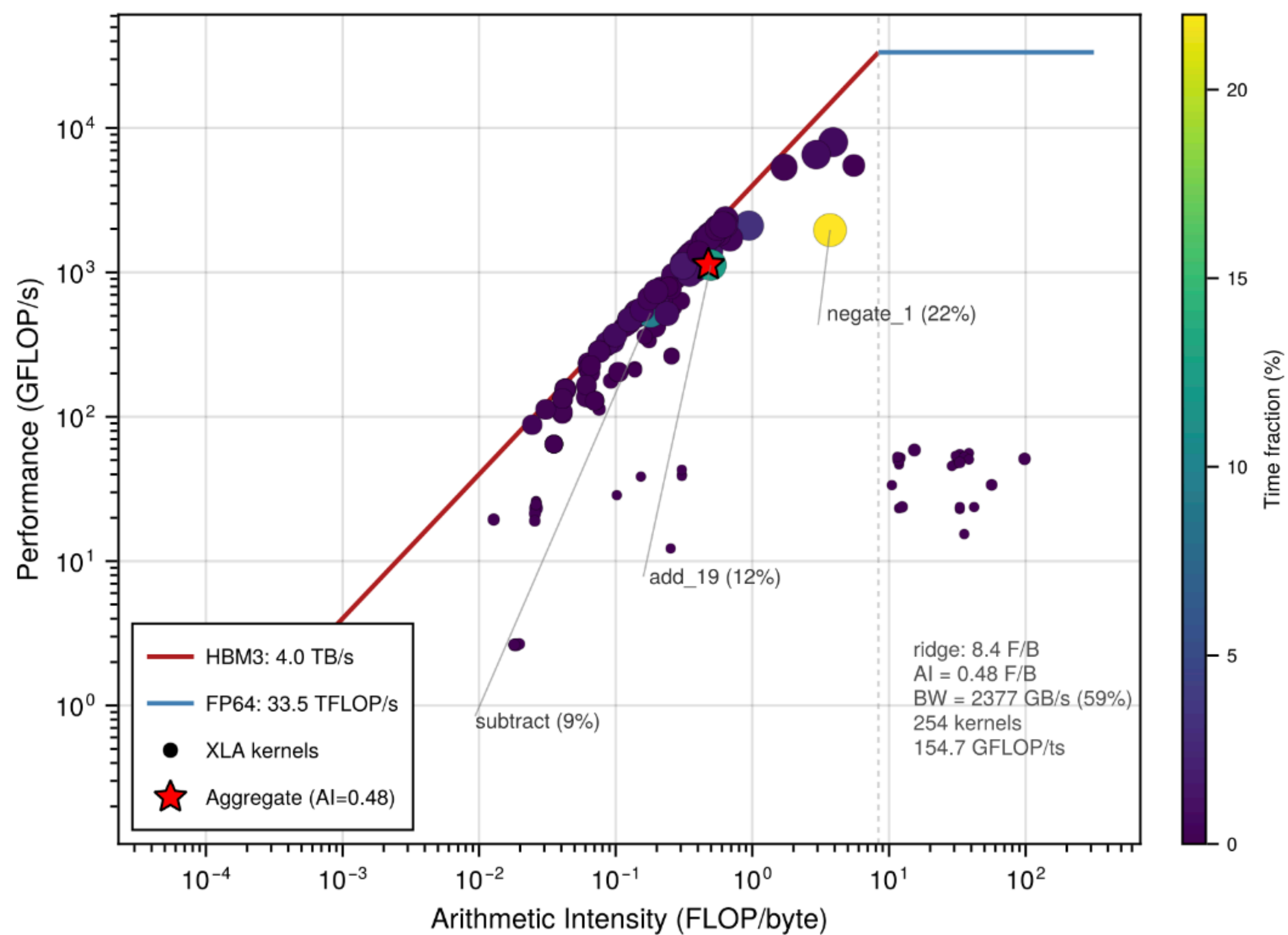
Google Cloud



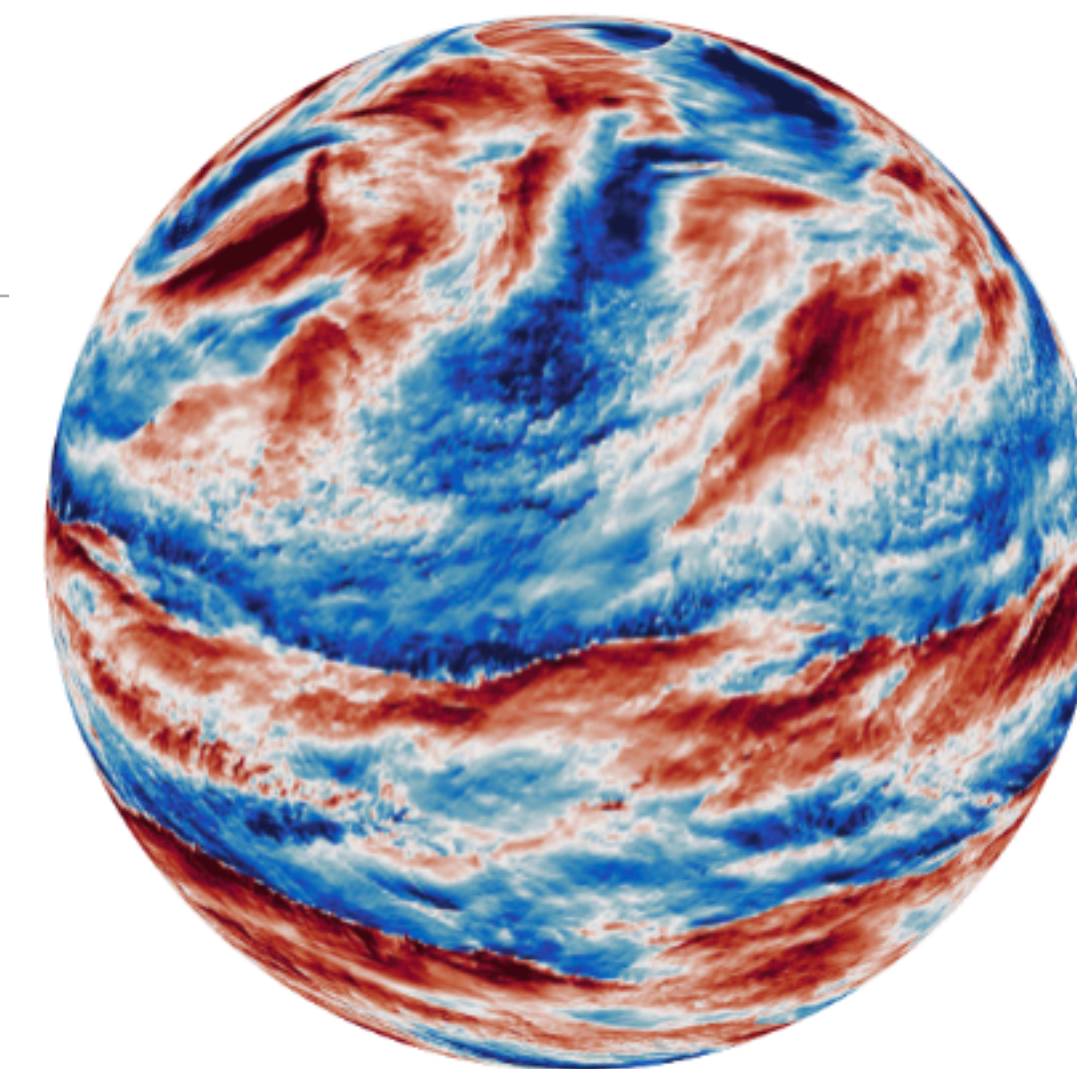
# Performance Results (Alps / NVIDIA GPU)



# Performance Results (Alps / NVIDIA GPU)

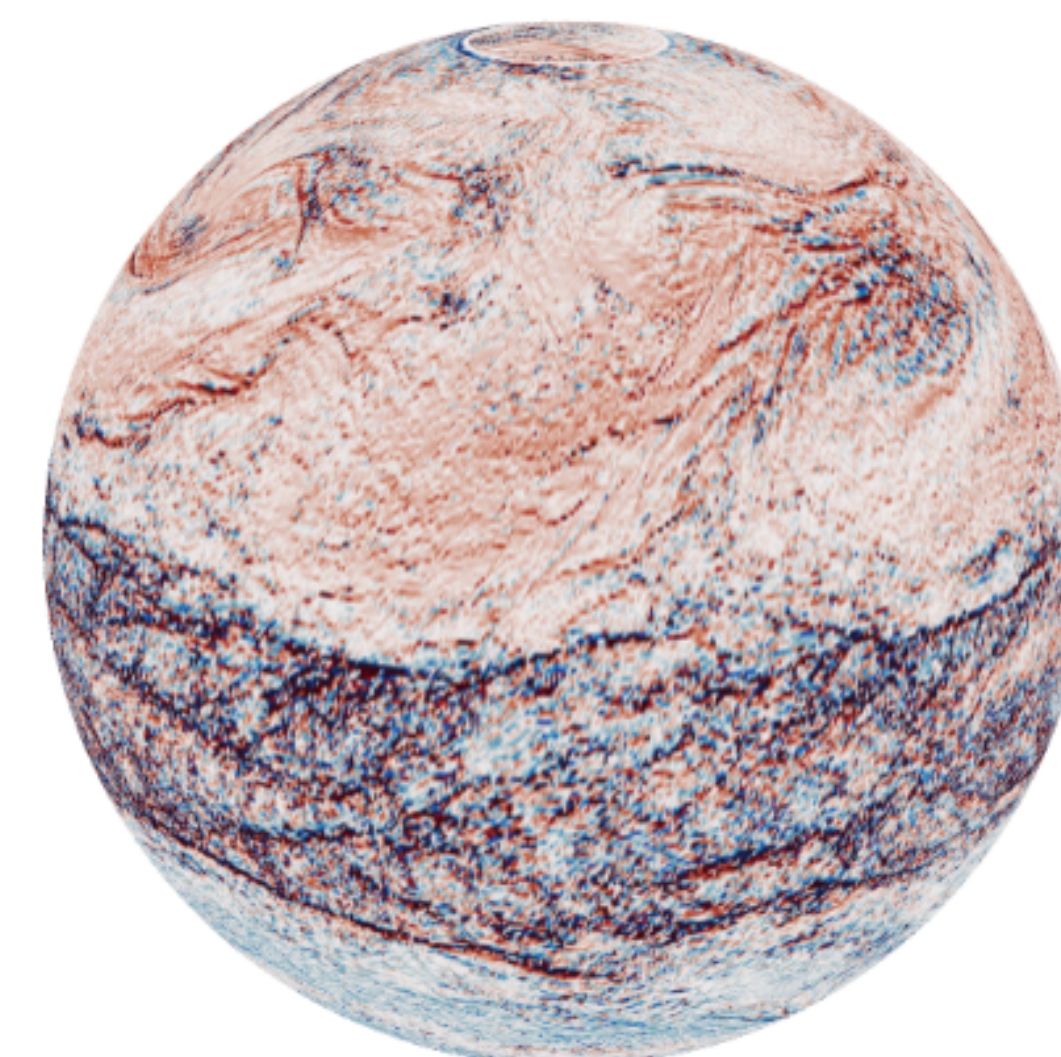


Meridional momentum,  $\rho v$



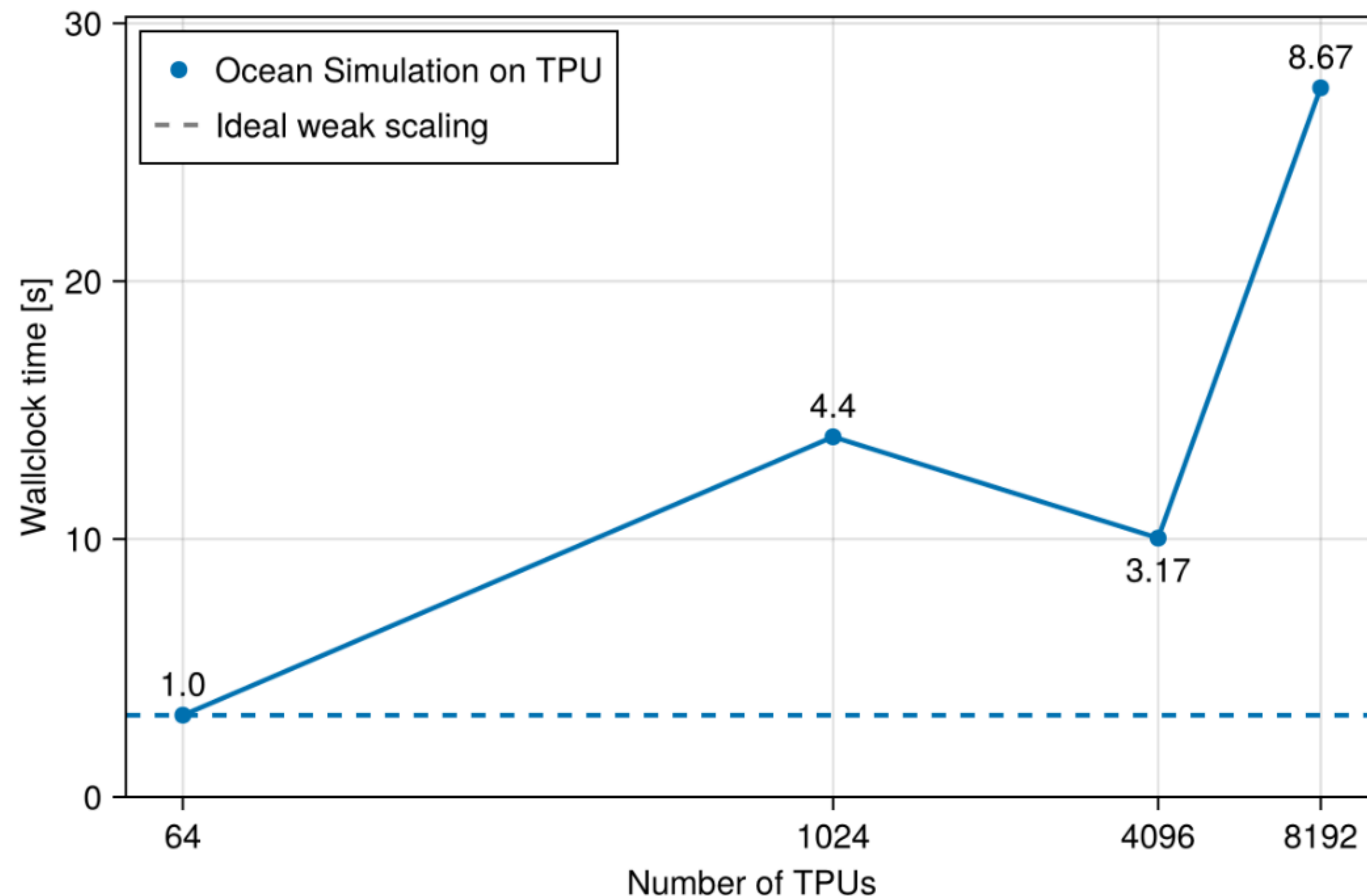
Sensitivity  $\partial J / \partial(\rho v_0)$

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



# 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%



# Enzyme Takeaways

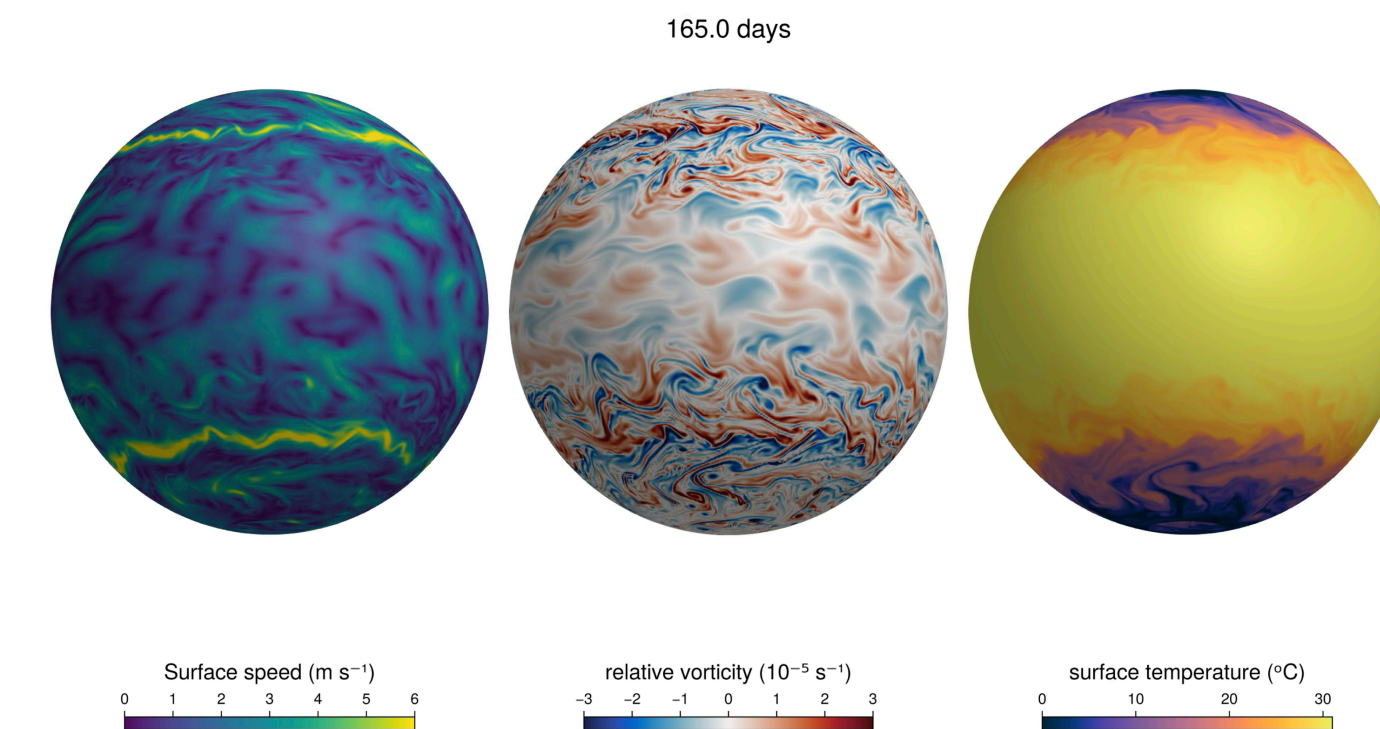
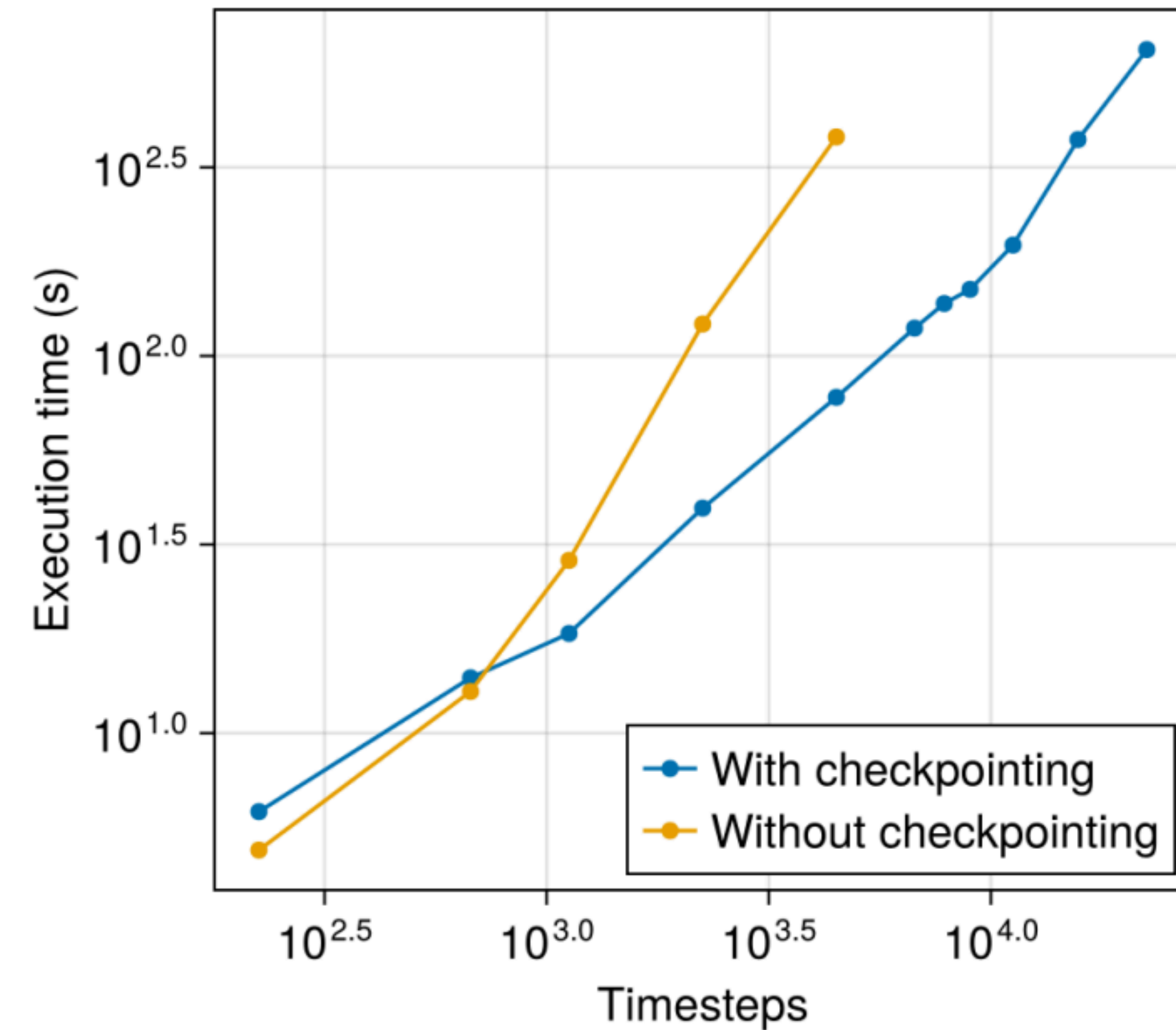
---

- Tool for performing reverse-mode AD of statically analyzable LLVM & MLIR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization on CPU, orders of magnitude improvements on GPU, and distributed
  - Interaction with Optimization is key!
- EnzymeMLIR preserves & optimizing custom semantics; impact compounds on accelerators
- Reactant{.jl} lets you take existing (C++, Julia, etc) code and run it on distributed AI chips
- All open source ([GitHub.com/EnzymeAD](https://github.com/EnzymeAD)/{Enzyme,Enzyme.jl,Enzyme-JaX, Reactant, ...} ; [enzyme.mit.edu](http://enzyme.mit.edu) ); weekly developer meetings!



# Derivative Raising Performance Results

- Primal Perf (CPU)
  - Vanilla Model: 272.0seconds
  - Tensor Optimis: 11.5seconds
- Derivative Performance
  - Similar performance to primal on single timestep, scaling with linearly time steps
  - Disabling tensor optimizations causes it to instantly oom the system
  - Tensor and whole-program optimizations are quite useful!



# Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix
- Without any optimization, we perform a scatter to create the diagonal, then a matmul

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)
```

```
v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))  
A = Reactant.to_rarray(rand(Float32, 1024, 1024))  
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:  
tensor<1024x1024xf32>) → tensor<f32> {  
  %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>  
  %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>  
  %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) →  
  tensor<1024x1024xf32>  
  %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>  
  %2 = "stablehlo.scatter"(%cst_0, %1, %arg0) <{scatter_dimension_numbers =  
#stablehlo.scatter<inserted_window_dims = [0, 1], scatter_dims_to_operand_dims = [0, 1],  
index_vector_dim = 1}>> ({  
  ^bb0(%arg3: tensor<f32>, %arg4: tensor<f32>):  
    stablehlo.return %arg4 : tensor<f32>  
  }) : (tensor<1024x1024xf32>, tensor<1024x2xi64>, tensor<1024xf32>) → tensor<1024x1024xf32>  
  %3 = stablehlo.dot_general %2, %arg1, contracting_dims = [1] x [1], precision = [DEFAULT,  
DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) → tensor<1024x1024xf32>  
  %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>  
  %5 = stablehlo.multiply %4, %4 : tensor<1024x1024xf32>  
  %6 = stablehlo.reduce(%5 init: %cst) applies stablehlo.add across dimensions = [0, 1] :  
(tensor<1024x1024xf32>, tensor<f32>) → tensor<f32>  
  return %6 : tensor<f32>  
}
```

# Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix
- Without any optimization, we perform a scatter to create the diagonal, then a matmul
- Differentiating this, results in gathers in the derivative, which cannot be removed via optimization.

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)
```

```
v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))  
A = Reactant.to_rarray(rand(Float32, 1024, 1024))  
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:  
tensor<1024x1024xf32>) → (tensor<1024xf32>, tensor<1024x1024xf32>, tensor<1024x1024xf32>)  
{  
  %cst = stablehlo.constant dense<-2.000000e+00> : tensor<1024x1024xf32>  
  %cst_0 = stablehlo.constant dense<2.000000e+00> : tensor<1024x1024xf32>  
  %cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>  
  %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) →  
tensor<1024x1024xf32>  
  %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>  
  %2 = stablehlo.broadcast_in_dim %arg0, dims = [0] : (tensor<1024xf32>) →  
tensor<1024x1024xf32>  
  %3 = stablehlo.multiply %2, %arg1 : tensor<1024x1024xf32>  
  %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>  
  %5 = stablehlo.multiply %4, %cst_0 : tensor<1024x1024xf32>  
  %6 = stablehlo.compare GE, %4, %cst_1 : (tensor<1024x1024xf32>, tensor<1024x1024xf32>)  
→ tensor<1024x1024xi1>  
  %7 = stablehlo.multiply %4, %cst : tensor<1024x1024xf32>  
  %8 = stablehlo.select %6, %5, %7 : tensor<1024x1024xi1>, tensor<1024x1024xf32>  
  %9 = stablehlo.transpose %8, dims = [1, 0] : (tensor<1024x1024xf32>) →  
tensor<1024x1024xf32>  
  %10 = stablehlo.dot_general %8, %arg1, contracting_dims = [1] x [0], precision =  
[DEFAULT, DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) →  
tensor<1024x1024xf32>  
  %11 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<1024xf32>) →  
tensor<1024x1024xf32>  
  %12 = stablehlo.multiply %8, %11 : tensor<1024x1024xf32>  
  %13 = "stablehlo.gather"(%10, %1) <{dimension_numbers =  
#stablehlo.gather<collapsed_slice_dims = [0, 1], start_index_map = [0, 1], index_vector_dim  
= 1>, slice_sizes = array<i64: 1, 1>> : (tensor<1024x1024xf32>, tensor<1024x2xi64>) →  
tensor<1024xf32>  
  return %13, %12, %9 : tensor<1024xf32>, tensor<1024x1024xf32>, tensor<1024x1024xf32>  
}
```

# Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix
- $\text{mul}(\text{diag}(x), v) \rightarrow \text{elementwise}(x, v)$
- Performing this prior to AD yields 2-3x performance!

## All Optimizations Enabled

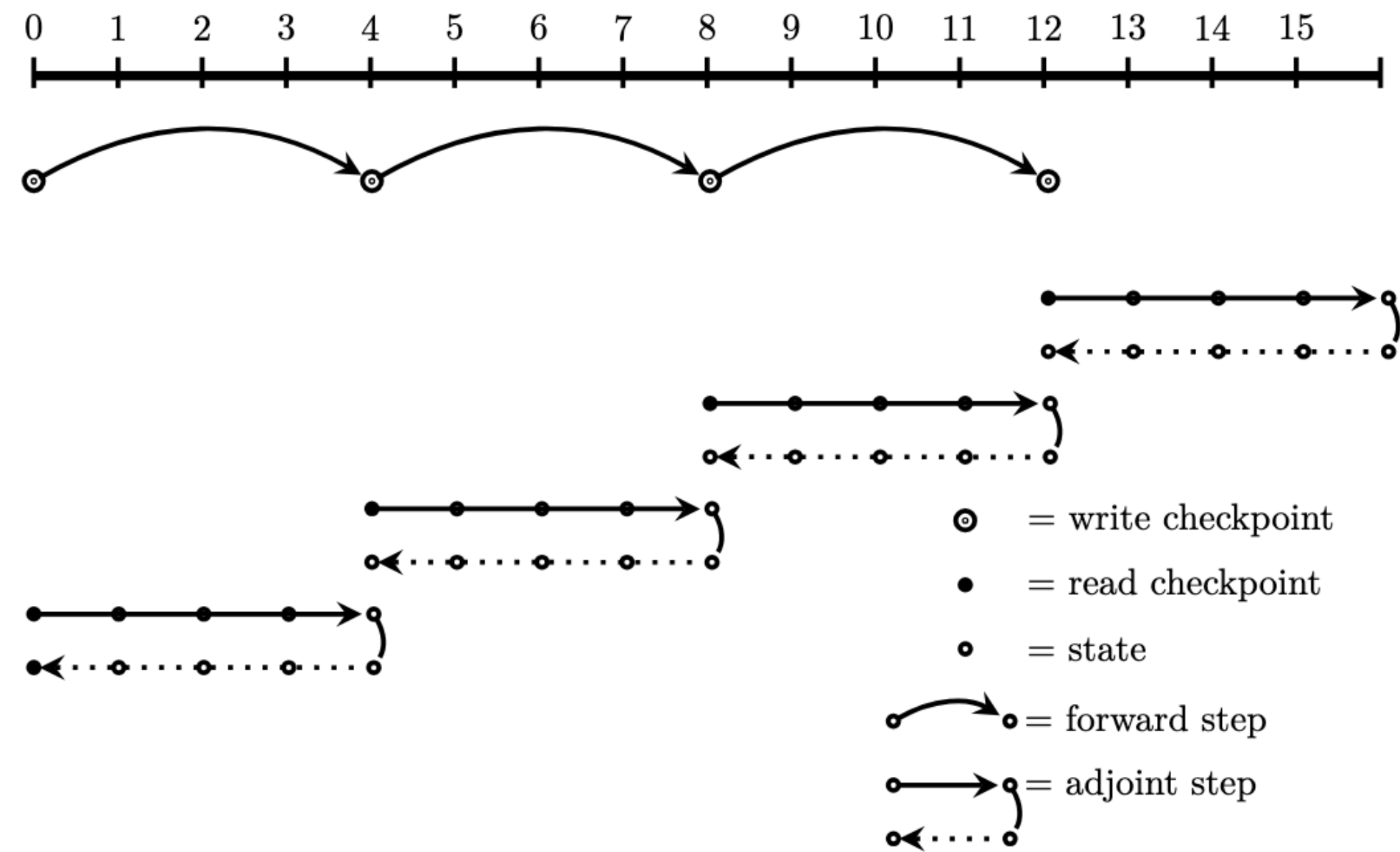
```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2: tensor<1024x1024xf32>)
→ tensor<f32> {
  %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
  %0 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<1024xf32>) → tensor<1024x1024xf32>
  %1 = stablehlo.multiply %0, %arg1 : tensor<1024x1024xf32>
  %2 = stablehlo.add %1, %arg2 : tensor<1024x1024xf32>
  %3 = stablehlo.multiply %2, %2 : tensor<1024x1024xf32>
  %4 = stablehlo.reduce(%3 init: %cst) applies stablehlo.add across dimensions = [0, 1] :
(tensor<1024x1024xf32>, tensor<f32>) → tensor<f32>
  return %4 : tensor<f32>
}
```

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) → tensor<f32> {
  %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
  %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>
  %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) →
tensor<1024x1024xf32>
  %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>
  %2 = "stablehlo.scatter"(%cst_0, %1, %arg0) <{scatter_dimension_numbers =
#stablehlo.scatter<inserted_window_dims = [0, 1], scatter_dims_to_operand_dims = [0, 1],
index_vector_dim = 1>}> ({
  ^bb0(%arg3: tensor<f32>, %arg4: tensor<f32>):
    stablehlo.return %arg4 : tensor<f32>
}) : (tensor<1024x1024xf32>, tensor<1024x2xi64>, tensor<1024xf32>) → tensor<1024x1024xf32>
  %3 = stablehlo.dot_general %2, %arg1, contracting_dims = [1] x [1], precision = [DEFAULT,
DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) → tensor<1024x1024xf32>
  %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>
  %5 = stablehlo.multiply %4, %4 : tensor<1024x1024xf32>
  %6 = stablehlo.reduce(%5 init: %cst) applies stablehlo.add across dimensions = [0, 1] :
(tensor<1024x1024xf32>, tensor<f32>) → tensor<f32>
  return %6 : tensor<f32>
}
```

## Scatter Optimizations Disabled

# Checkpointing

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

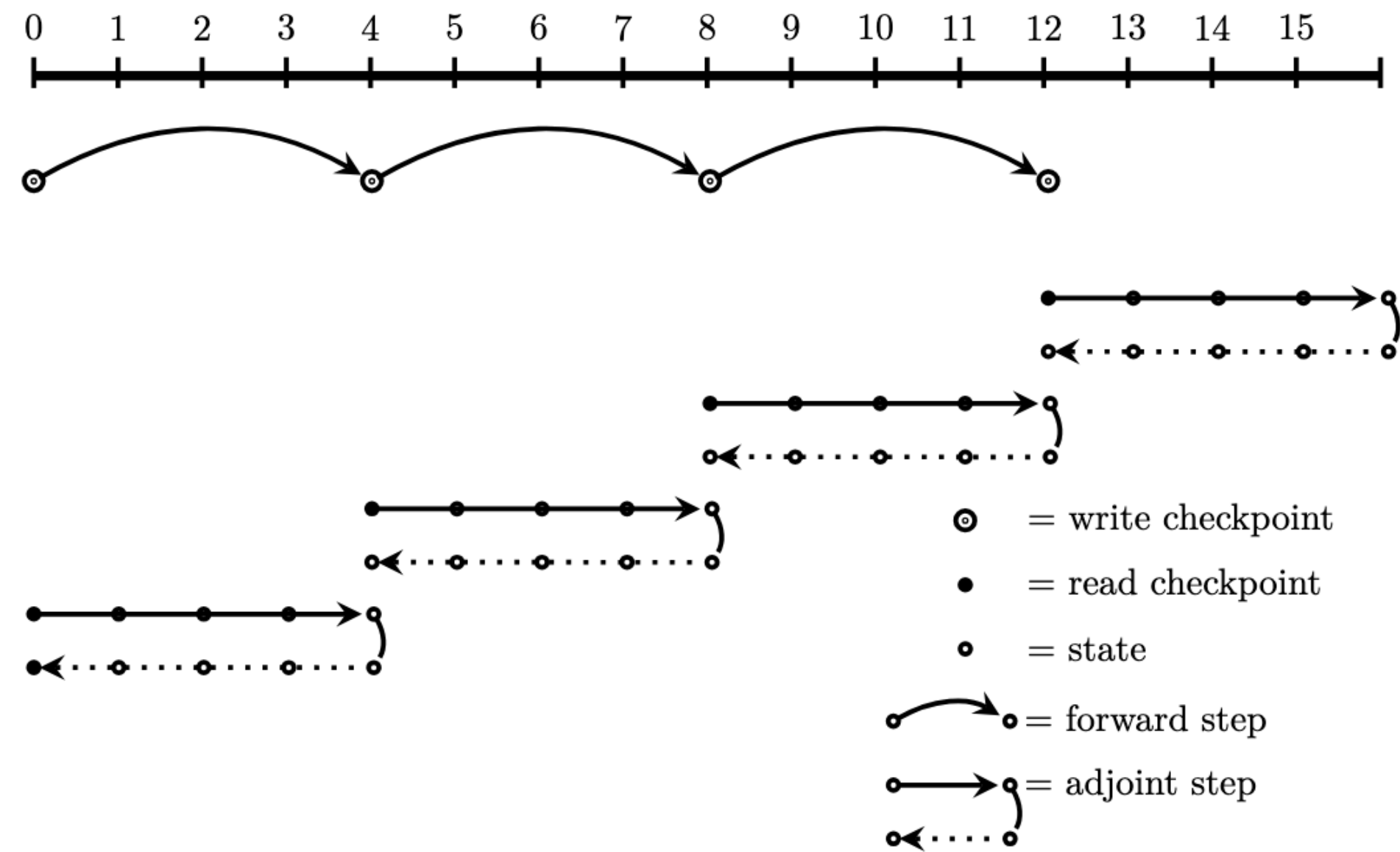


```
cache = malloc N x f32
for i = 0:N {
    x = foo(x)
    cache[i] = x
}

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

# Checkpointing

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



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

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

# Checkpointing

- Checkpointing is a technique for trading off memory and compute time in the derivative
- Performing entire-program-level analysis, we can remove induction variables on the loop, reducing memory AND computation

```
x = tensor<100x100xf32>
for i = 0:steps {
    x[0, :] = 0
    x[end, :] = 0
    y = foo(x, y)
}
```

```
if (steps > 0) {
    x[0, :] = 0
    x[end, :] = 0

    for i = 0:steps {
        y = foo(x, y)
    }
}
```

# Linear Algebra + AD

---

- Consider a simple code which performs a matmul and add on a Diagonal matrix

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)
```

```
v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))  
A = Reactant.to_rarray(rand(Float32, 1024, 1024))  
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

# 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  
(Clang Plugin for C++, JIT Package for Julia, etc)
- 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  
}
```

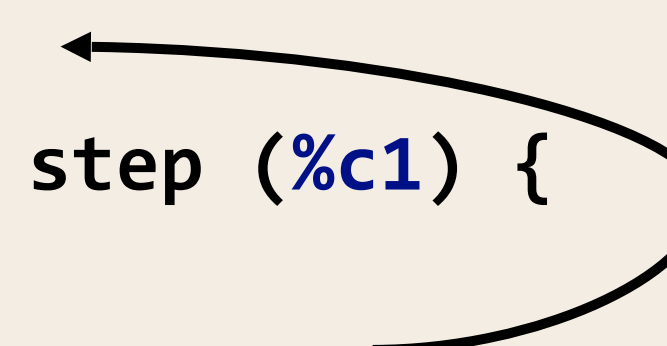


# GPU Programming via MLIR

- Preserve Host & Device code through frontend  
(Clang Plugin for C++, JIT Package for Julia, etc)
- 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

---

```
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.alloc : 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
- Another



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

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

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

Jens Domke  
jens.domke@riken.jp  
RIKEN  
Japan

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

Johannes Doerfert  
jdoerfert@llnl.gov  
LLNL  
United States

Oleksandr Zinenko  
zinenko@google.com  
Google  
France

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

We propose an alternative approach that automatically translates programs written in one programming model (CUDA), into another (CPU threads) based on Polygeist/MLIR. Our approach includes a representation of parallel constructs that allows conventional compiler transformations to apply transparently and without modification and enables parallelism-specific optimizations. We evaluate our framework by transpiling and optimizing the CUDA Rodinia benchmark suite for a multi-core CPU and achieve a 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>

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

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

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU translation has been explored in the past [9, 23, 58], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transforms, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [39]. As representations of parallelism within a mainstream compiler have only recently begun to

- 27% speedup on real code, 2.7x on PyTorch cross compilation!

## Retargeting and Respecializing GPU Workloads for Performance Portability

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

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

```
    // Necessary Barrier #1  
    // of the read/writes below the sync  
    // (ights, hidden)  
    // intersect with the read/writes above the sync  
    // (de, input)  
    threads();
```

```
    // Necessary 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);  
  }  
}
```

# Allocation Merging

---

- Allocations (and any calls) on the GPU are expensive
- Given two allocations in the same scope, replace uses with a single allocation
- Beneficial for not just AD, but any GPU programs!

```
double* var1 = new double[N];
double* var2 = new double[M];

use(var1, var2);

delete[] var1;
delete[] var2;
```

```
double* var1 = new double[N + M];
double* var2 = var1 + N;

use(var1, var2);

delete[] var1;
```

