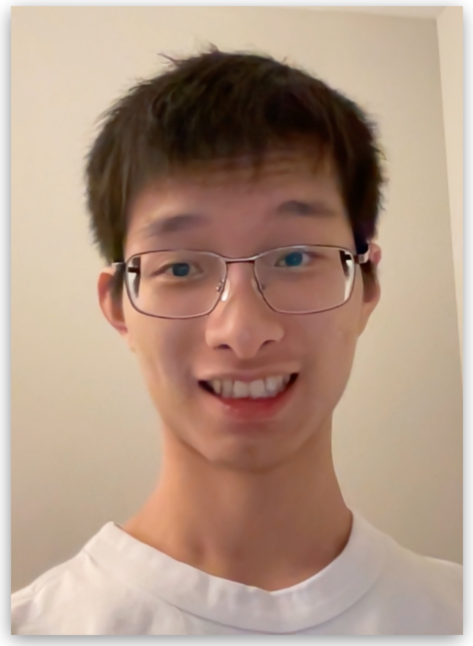
AD-aware Compiler Optimizations in Enzyme MLIR

Vimarsh Sathia[1] Siyuan Brant Qian[1] Jan Hückelheim[2] Paul Hovland[2] William S. Moses[1]

vsathia2@illinois.edu

[1] University of Illinois Urbana-Champaign
[2] Argonne National Laboratory

MFEM Community Workshop, Sep 10, 2025

William S. Moses[†,§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, Paul Berg[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], Son Vu, Sergio Sanchez-Ramirez[◊], Simone Silvestri, Nora Loose[♣], Ivan Ho, Vimarsh Sathia[†], Jan Hueckelheim[♠], Johannes De Fine Licht, Kevin Gleason[§], Ludovic Rass, Gabriel Baraldi, Dhruv Apte[#], Lorenzo Chelini[♦], Jacques Pienaar[§], Gaetan Lounes, Valentin Churavy, Sri Hari Krishna Narayanan[♠], Navid Constantinou, William R. Magro[§], Michel Schanen[♠], Alexis Montoison[♠], Alan Edelman[‡], Samarth Narang, Tobias Grosser, Keno Fischer[♮], Robert Hundt[§], Albert Cohen[§], Oleksandr Zinenko[§] [*]

UIUC [†], Google [§], UCL [★], MIT [‡], NVIDIA [♦], UT Austin [#], [C]Worthy [♣], BSC [◊], Argonne National Laboratory [♠], LBNL [♡], Cambridge [♭], JuliaHub [♮], University of Mainz [♯], BFH [▽], Ghent University [△]

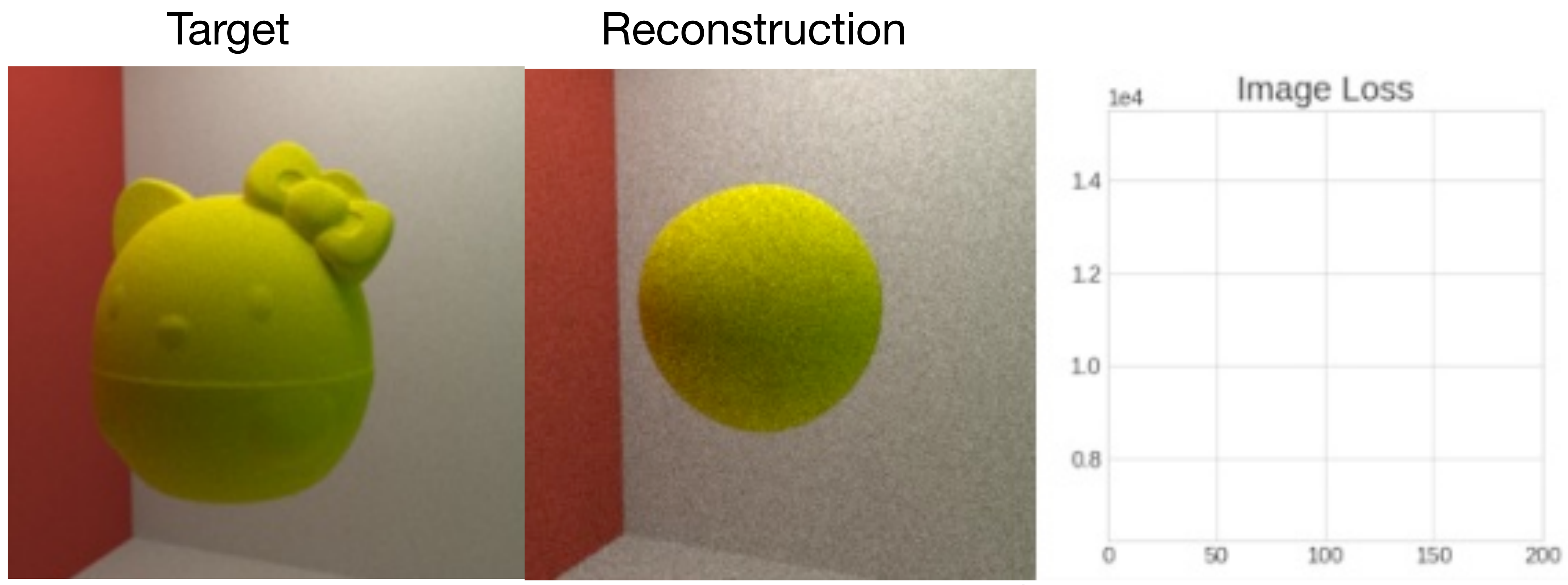
Outline

- Compiler-Based Differentiation (Enzyme-LLVM)
- What is Enzyme-MLIR?
- Case Study: Tensor Algebra Optimization
- Case Study: Higher Order Derivatives
- AD-Specific Optimizations

Differentiation: Connecting Science and AI

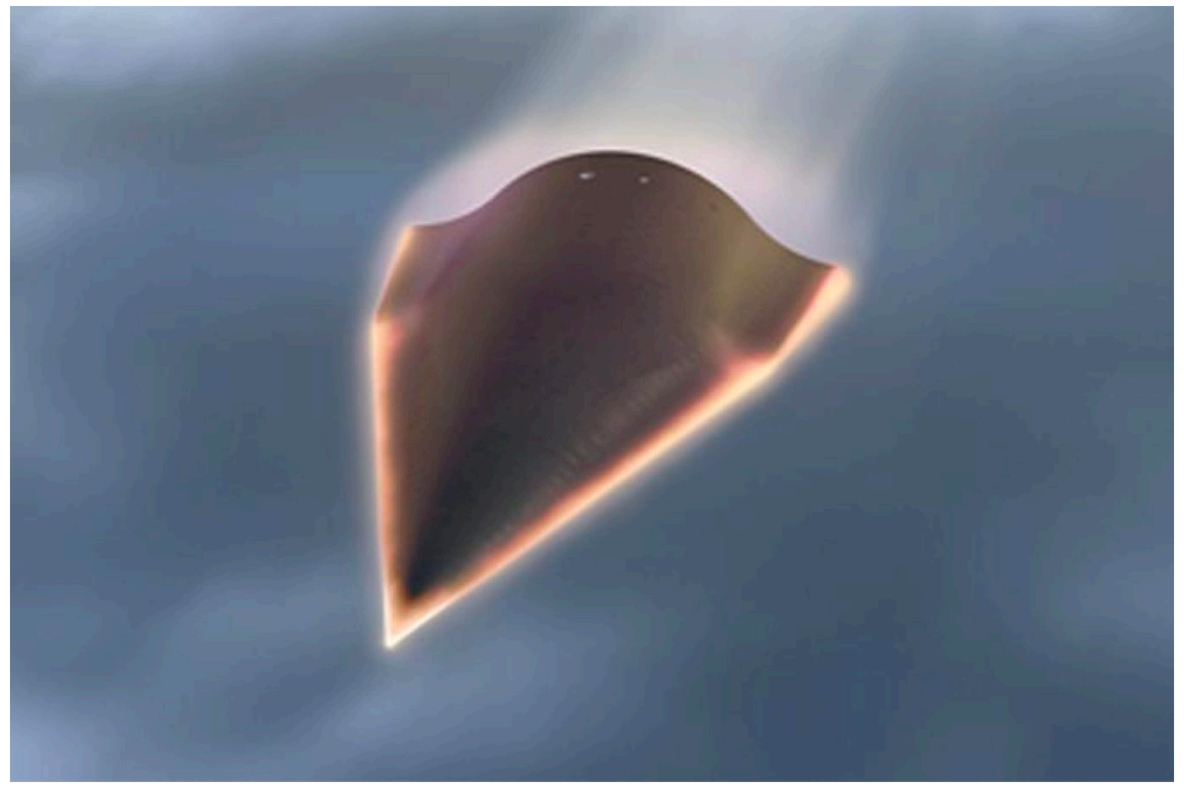
Derivatives are key to science + ML

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

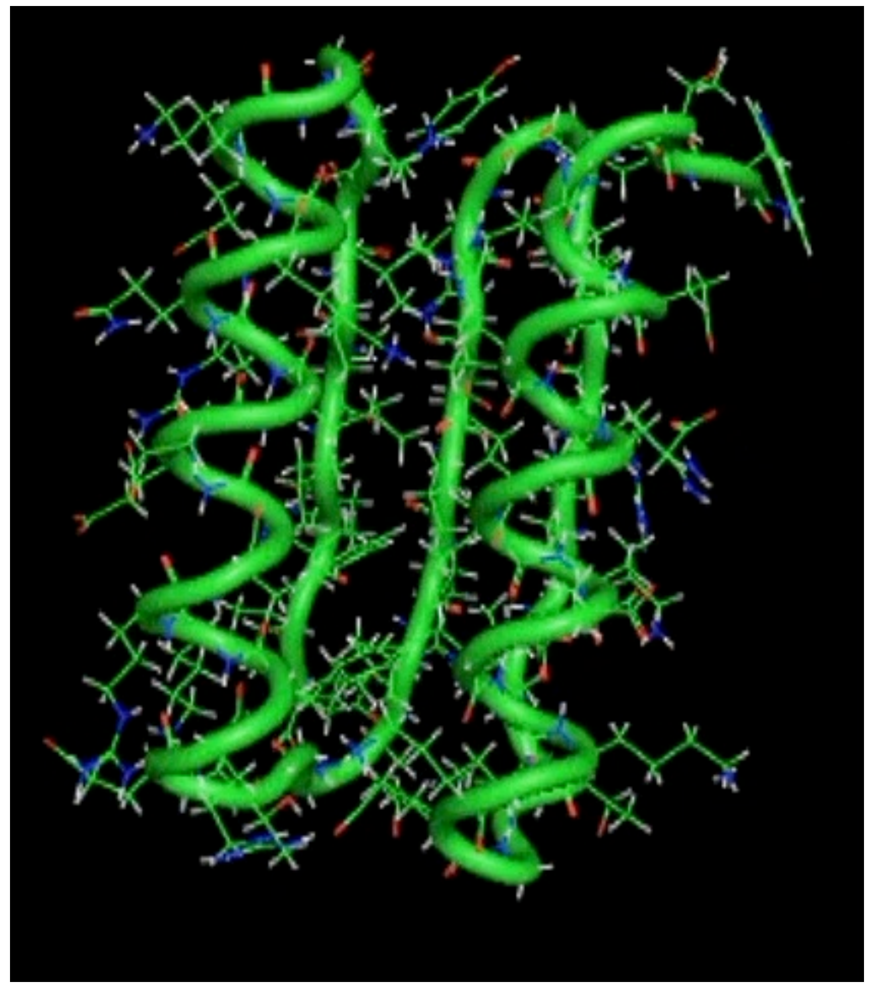


from Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering, SIGGRAPH Asia 2022, Zihan Yu et al

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;
}
```

AD

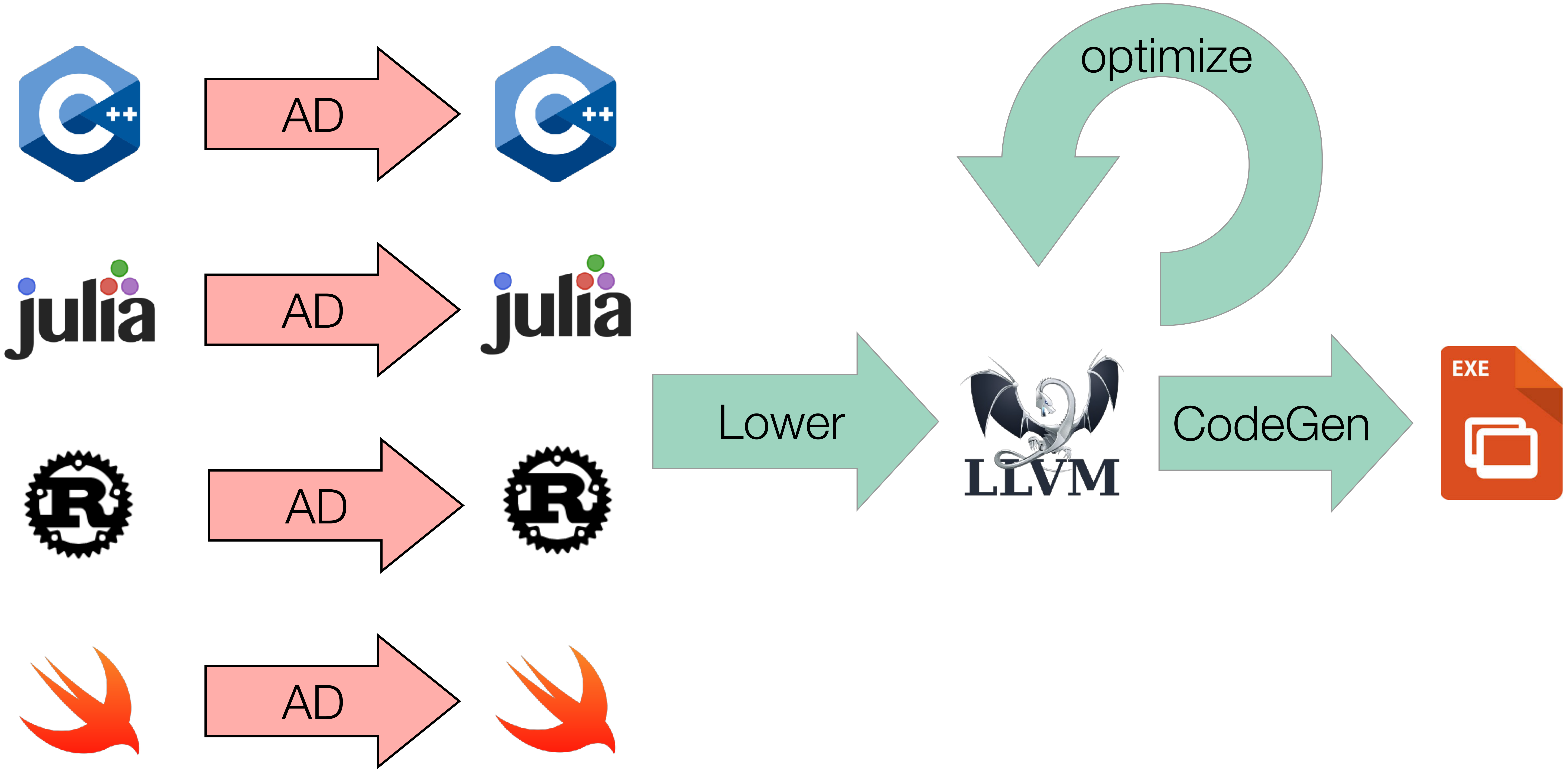
```
double grad_relu3(double x) {
  if (x > 0)
    return 3 * pow(x,2)
  else
    return 0;
}
```

- 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[i] = input[i];
  input2[i] += 0.001;
  grad_input[i] = (f(input2) - f(input))/0.001;
}
```

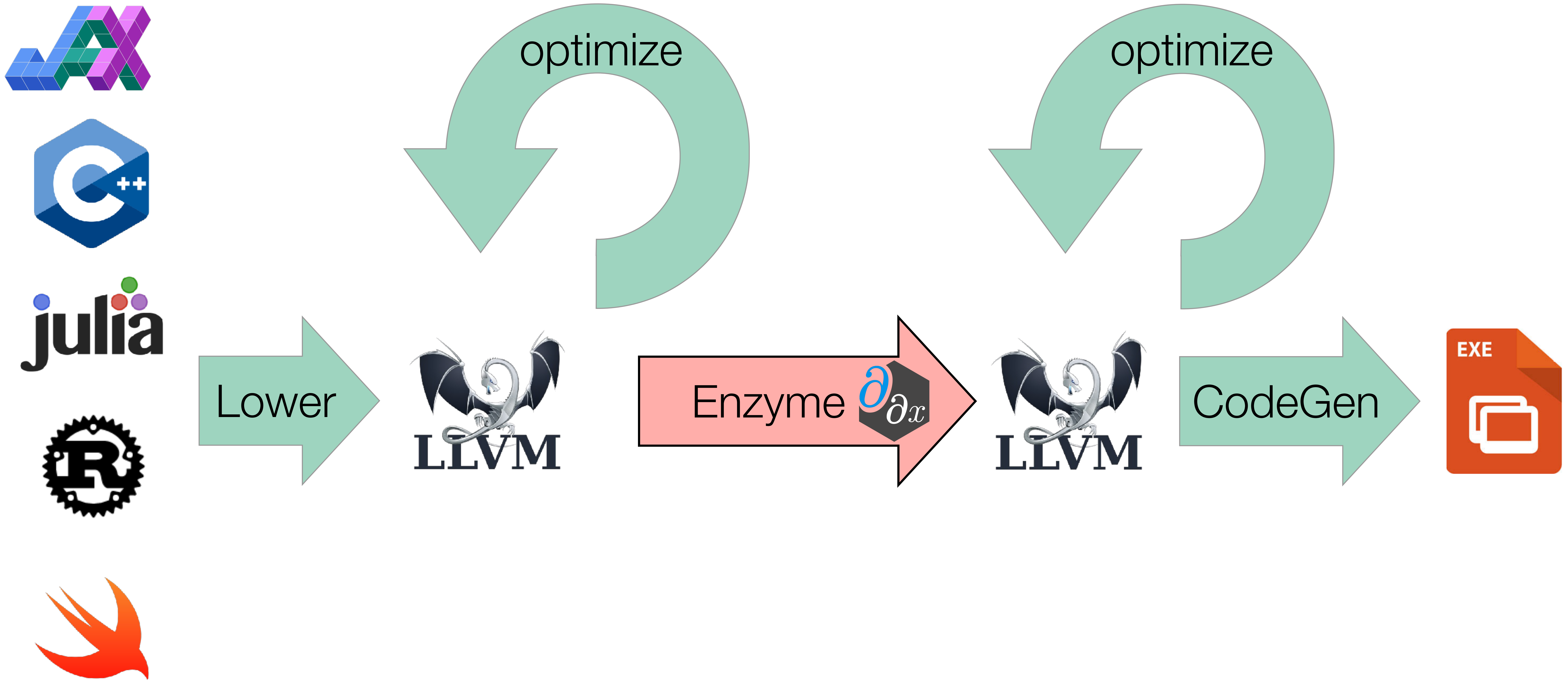
```
// Automatic differentiation
double grad_input[100];

grad_f(input, grad_input)
```

Existing Automatic Differentiation Pipelines



Enzyme Approach - Compiler Based Differentiation

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



What is MLIR?

Multi-Level Intermediate Representation (MLIR)

- New Compiler IR with user-defined dialects, instructions, optimizations
  - Arithmetic(`arith`), Linear Algebra(`linalg`), Complex Numbers(`complex`)
  - GPU Programming(`gpu`), Control Flow(`scf`)
  - Automatic Differentiation(`EnzymeMLIR`)

```
func @set(%arr: memref<?xf32>, %val: f32) -> f32 {
    scf.for %ii = 0 to 10 {
      memref.store %val, %arr [2 * %ii] : memref<?xf32>
    }
    %out = arith.mulf %val, %val : f32
    return %out
  }
}
```

Multi-Level Intermediate Representation (MLIR)

- New Compiler IR with user-defined dialects, instructions, optimizations
  - Arithmetic(`arith`), Linear Algebra(`linalg`), Complex Numbers(`complex`)
  - GPU Programming(`gpu`), Control Flow(`scf`)
  - Automatic Differentiation(`EnzymeMLIR`)
- **Mix and match operations and operands across multiple dialects** ✪
- Core infrastructure of modern ML frameworks (JaX, PyTorch, TensorFlow)

```
✪func @set(%arr: memref<?xf32>, %val: f32) -> f32 {
  ✪scf.for %ii = 0 to 10 {
    ✪memref.store %val, %arr [2 * %ii] : memref<?xf32>
    }
    %out = arith.✪mulf %val, %val : f32
    return %out
  }
}
```

Multi-Level Intermediate Representation (MLIR)

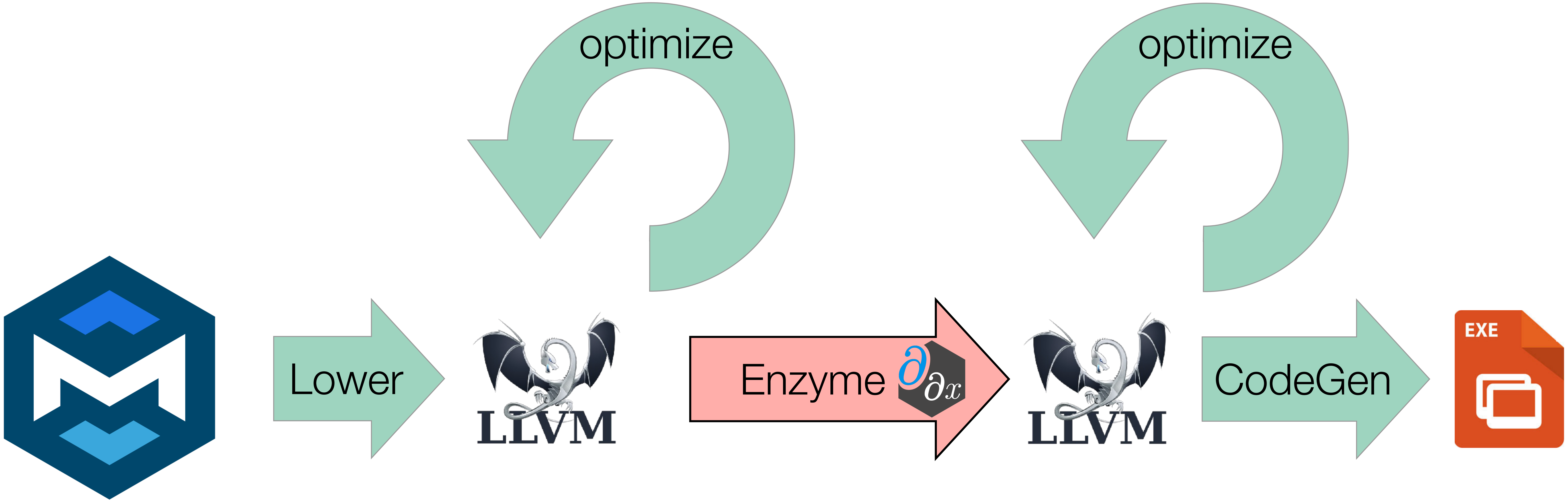
- New Compiler IR with user-defined dialects, instructions, optimizations
  - Arithmetic(`arith`), Linear Algebra(`linalg`), Complex Numbers(`complex`)
  - GPU Programming(`gpu`), Control Flow(`scf`)
  - Automatic Differentiation(`EnzymeMLIR`)
- Mix and match dialects and optimizations from multiple dialects
- Core infrastructure of modern ML frameworks (JaX, PyTorch, TensorFlow)

```
func @set(%arr: memref<?xf32>, %val: f32) -> f32 {
    scf.for %ii = 0 to 10 {
      memref.store %val, %arr [2 * %ii] : memref<?xf32>
    }
    %out = arith.mulf %val, %val : f32
    return %out
  }
}
```

EnzymeMLIR autodiff

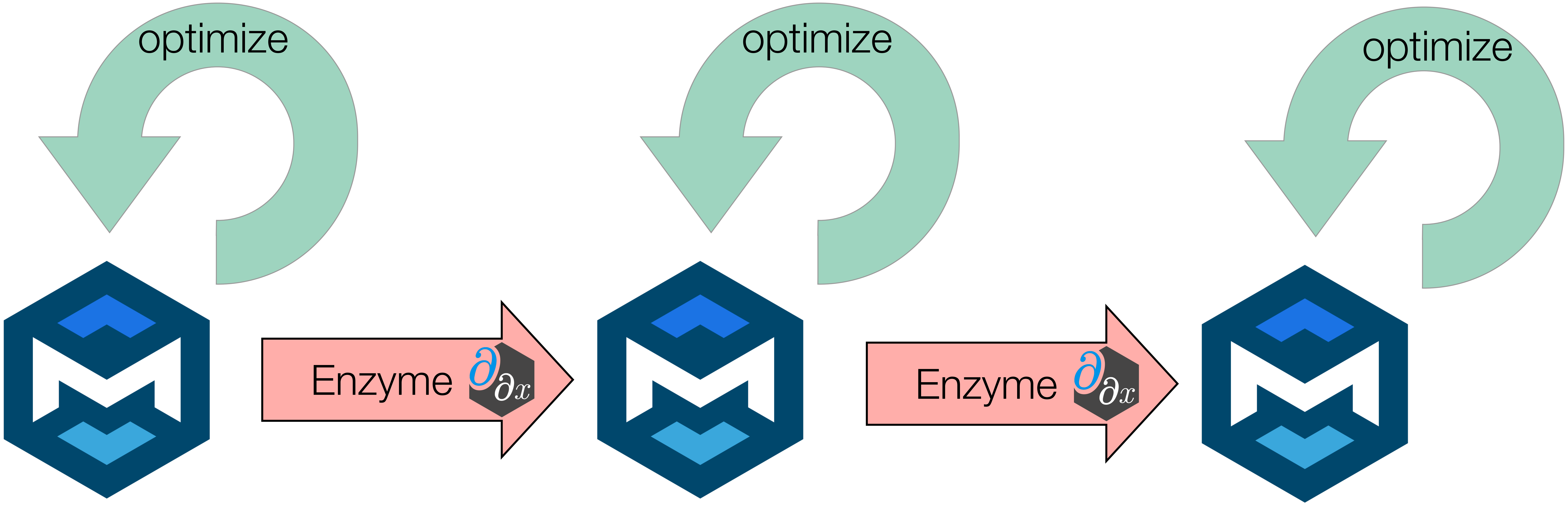
```
func @grad_set(%X: memref<?xf32>, %v: f32, %dout: f32) {
 %out, %dv = enzyme.autodiff @set(%X,%v,%dout) {
    activity = [enzyme_const, enzyme_active]
    ret_activity = [enzyme_active]
 } : (f32, f32)
 return
}
```

Why Enzyme-MLIR?



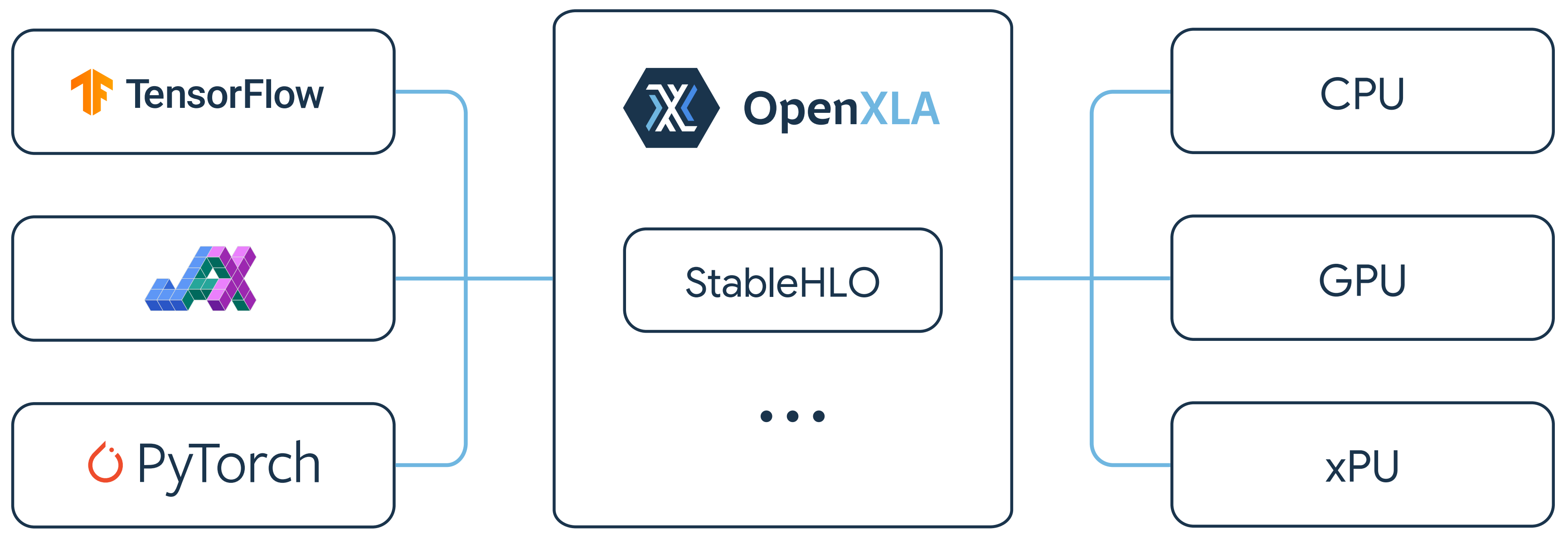
Why Enzyme-MLIR?

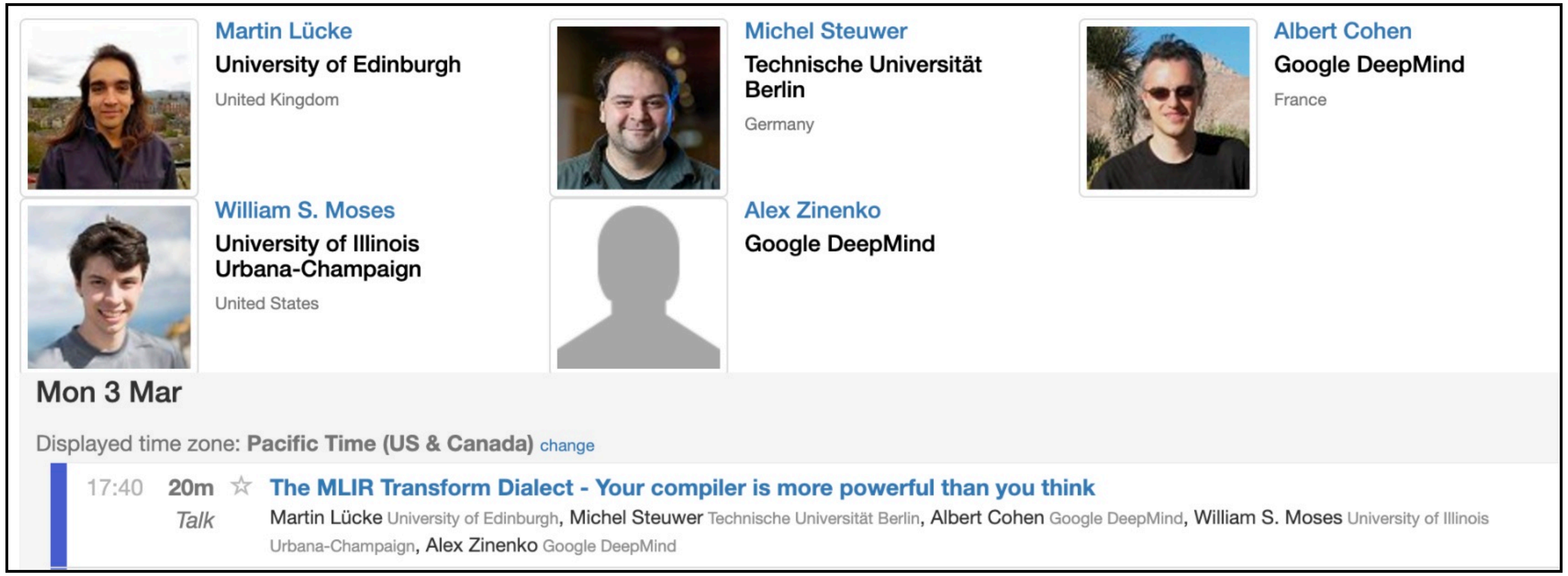
“Multi-level” coordination of AD and Optimization!



Case Study: Tensor Algebra Optimization

- `stablehlo` is a MLIR dialect which represents tensor algebra operations
- Implemented 200+ tensor rewrite rules to optimize code!
- **Hypothesis**: Optimizations on primal => ***outsized impact for derivatives***





CGO 2025 paper

Tensor Algebra Optimization: example

- `stablehlo` is a MLIR dialect which represents tensor algebra operations

- Implemented 200+ tensor rewrite rules to optimize code!

- **Hypothesis**: Optimizations on primal => ***outsized impact for derivatives***

```
// Some example rules
 x + 0 -> x
 transpose(transpose(x)) -> x

// push slices up(reduce work)
 slice(add(a,b)) -> add(slice(a),slice(b))

// push pads down(reduce work)
 mul(pad(x,0),y) -> pad(mul(x,slice(y)),0)
```

```
x,y = tensor<10000xf32>
a = dot(x,y)
b = mul(a,z)
c = dot(b[0:10],4)
return c;
```

Tensor Algebra Optimization: reduce matmul size

- `stablehlo` is a MLIR dialect which represents tensor algebra operations
- Implemented 200+ tensor rewrite rules to optimize code!
- **Hypothesis**: Optimizations on primal => ***outsized impact for derivatives***

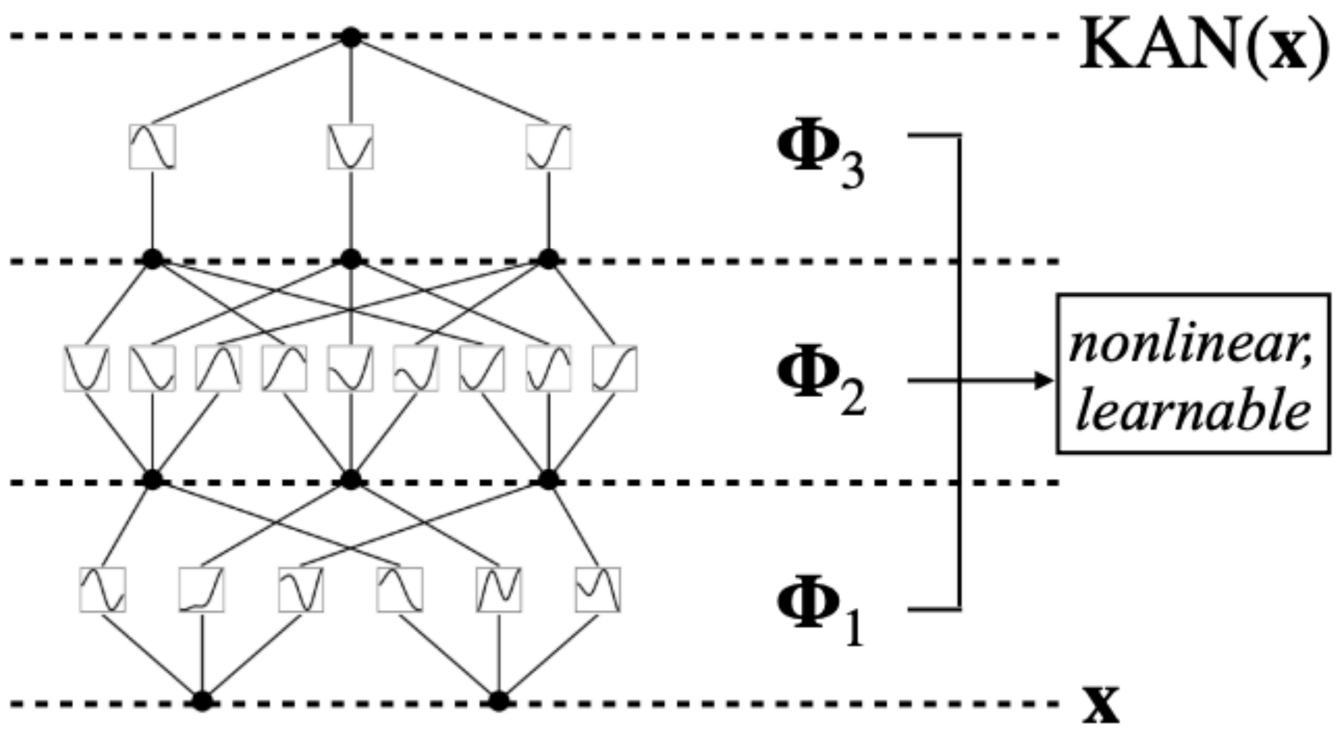
```
// Some example rules
 x + 0 -> x
 transpose(transpose(x)) -> x

// push slices up(reduce work)
 slice(add(a,b)) -> add(slice(a),slice(b))

// push pads down(reduce work)
 mul(pad(x,0),y) -> pad(mul(x,slice(y)),0)
```

```
x,y = tensor<10000xf32>
a = dot(x,y)
b = mul(a,z)
c = dot(b[0:10],4)
return c;
```

```
x,y = tensor<10000xf32>
a = dot(x,y)
b = mul(a[0:10],z[0:10])
c = dot(b,4)
return c;
```

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)
```

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

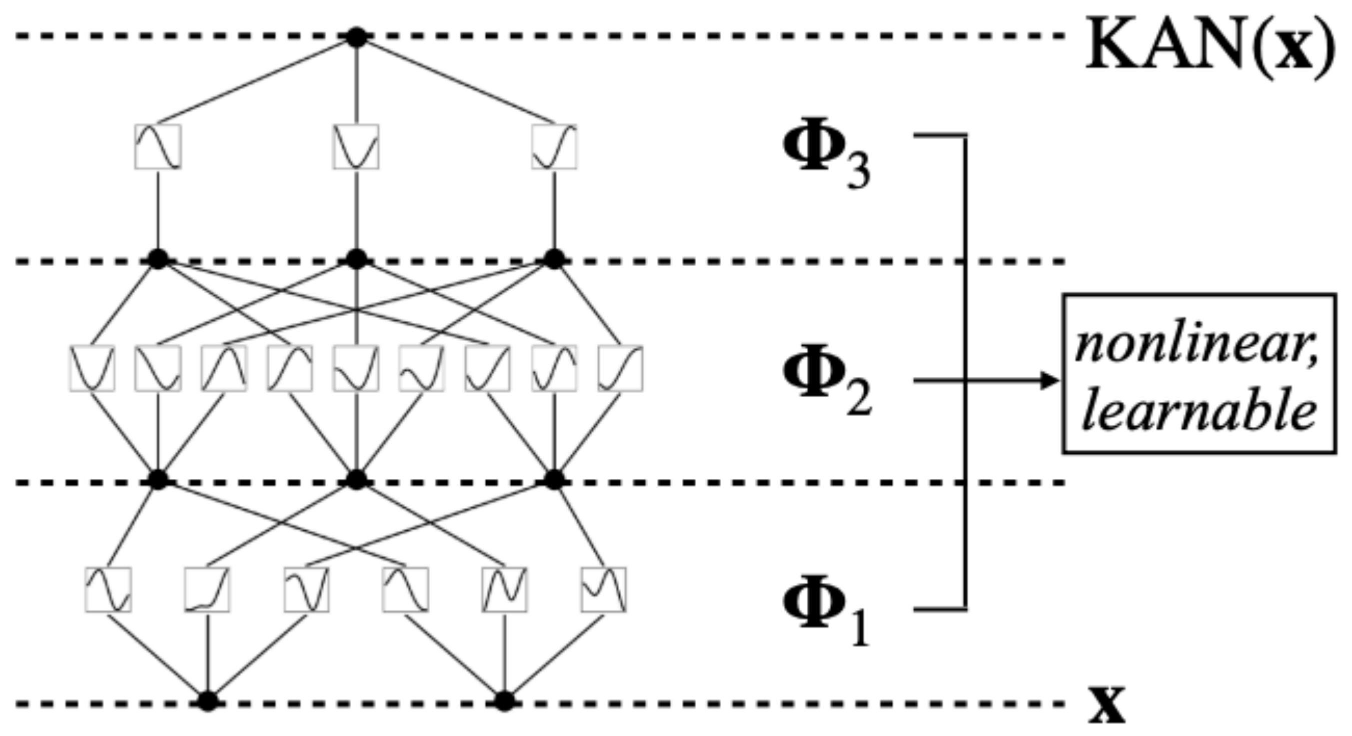
```
Backwards (Zygote + Julia)
 289.319 us ( 575 allocations)
2099.000 us (1055 allocations)
1772.000 us ( 877 allocations)
```

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

2.14x speedup
(Primal)

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)
```

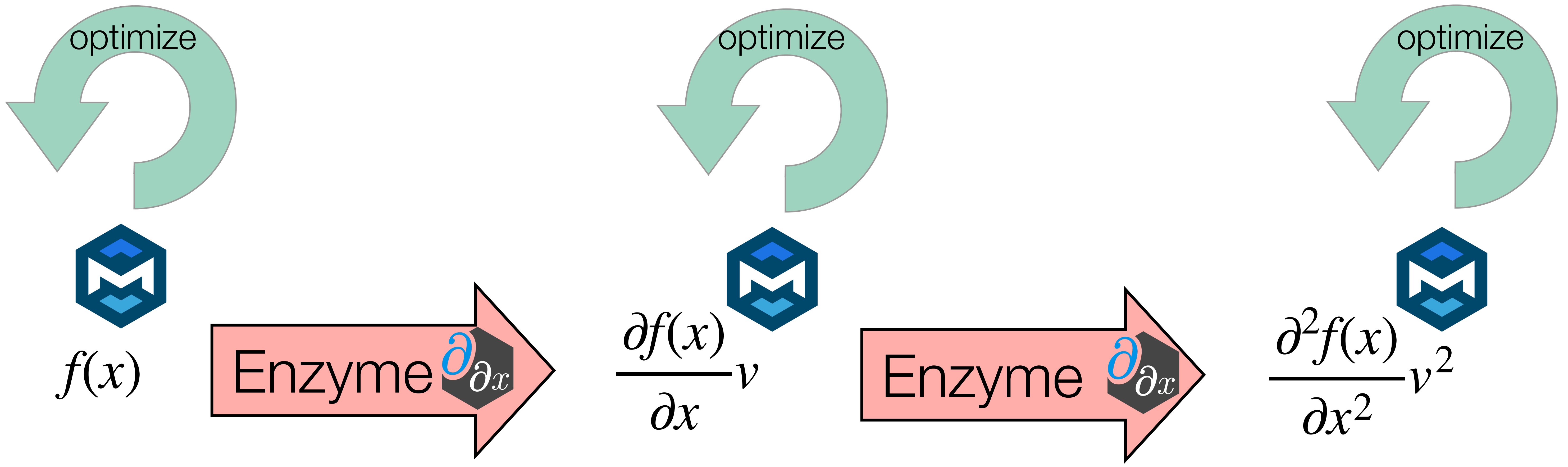
```
Forward (Reactant)
39.873 us (  2 allocations)
68.439 us (  6 allocations)
55.889 us (  6 allocations)
```

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

2.14x speedup (Primal) → 13.57x speedup (Derivative)

Case Study: Higher Order Differentiation

- Mathematical structure in higher-order derivatives (like symmetry, sparsity) leaves significant room for perf engineering
- Progressively running optimizations ***during AD*** helps make computations tractable.



symmetric derivatives

$$\frac{\partial^2 f}{\partial x\ \partial y} = \frac{\partial^2 f}{\partial y\ \partial x}$$

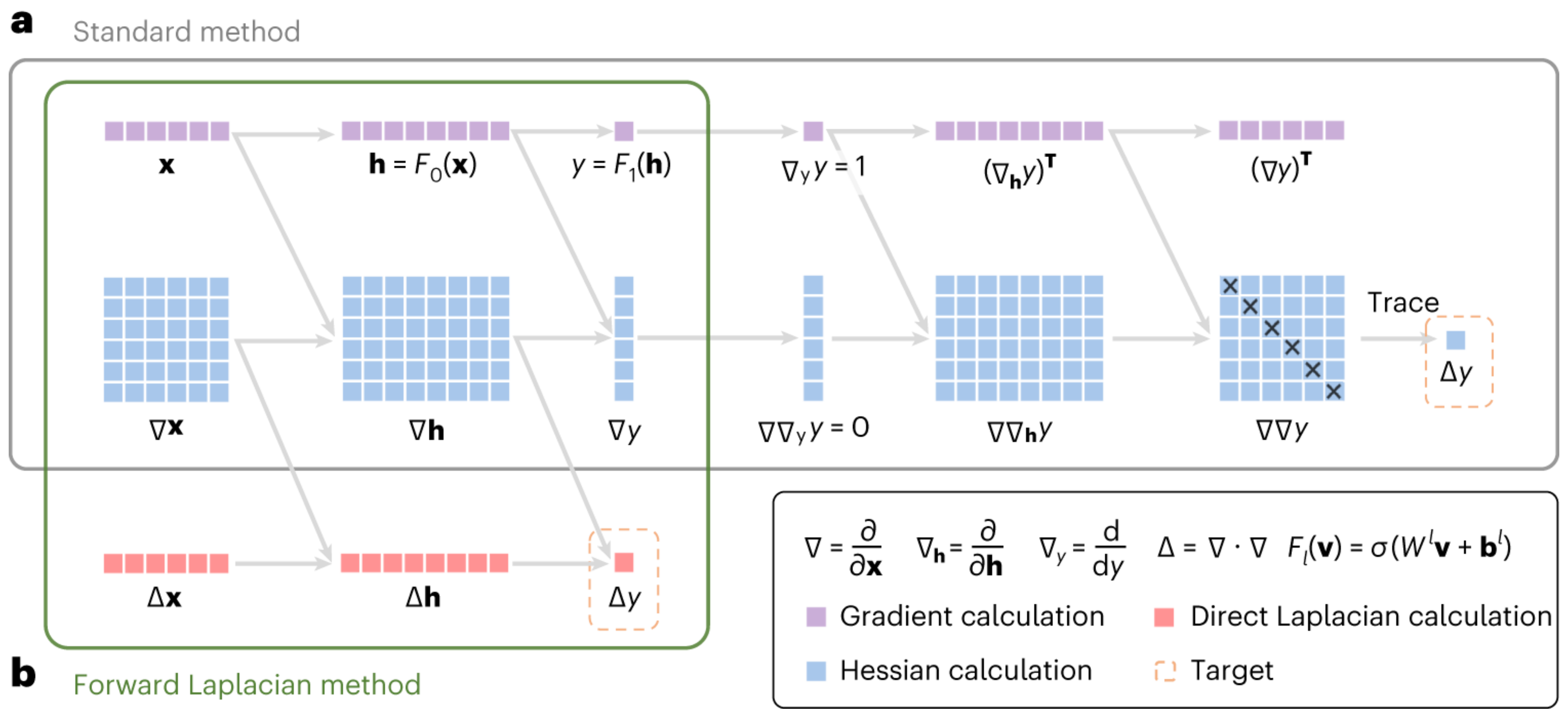
sparse Hessians

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$H(\vec{x}) = \begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

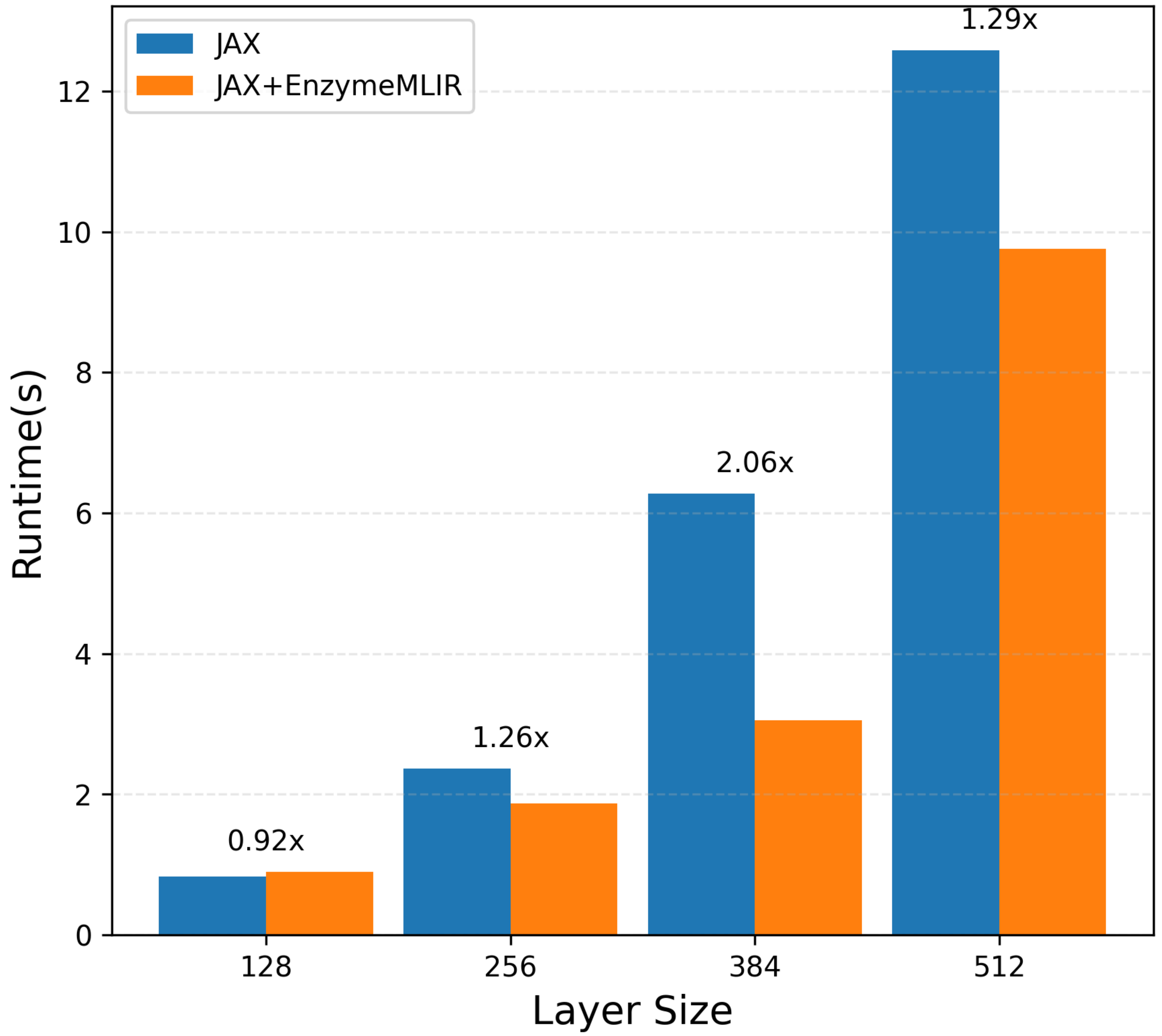
EnzymeMLIR in Python (via JAX MLIR frontend)

CPU Laplacian of Neural Net(NN) used in NN-based VMC



Laplace operator

$$\nabla^2 f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2}$$



1.324x speedup (Forward Laplacian)

Ongoing Work

+

Activity in EnzymeMLIR

- Enzyme attaches an activity attribute(`enzyme_const`, `enzyme_active`) to each input and output of function we want to differentiate

- Activities dictate how an `enzyme.autodiff` is lowered into gradient MLIR code.

- Optimizing activity assignment => ***optimizing generated derivative code***

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active, enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o1,%o2,%dx,%dy : (f32,f32,f32,f32)
}
```

Activity in EnzymeMLIR

- Enzyme attaches an activity attribute(`enzyme_const`, `enzyme_active`) to each input and output of function we want to differentiate
- Activities dictate how an `enzyme.autodiff` is lowered into gradient MLIR code.
- Optimizing activity assignment => ***optimizing generated derivative code***

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active, enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o1,%o2,%dx,%dy : (f32,f32,f32,f32)
}
```

EnzymeMLIR lowering

```
func @grad_square(%arg0: f32, %arg1: f32, %arg2: f32, %arg3: f32) {
  %0:4 = call @diffesquare(%arg0, %arg1, %arg2, %arg3)
  return %0#0, %0#1, %0#2, %0#3 : f32, f32, f32, f32
}

func private @diffesquare(%arg0: f32, %arg1: f32, %arg2: f32, %arg3: f32) -> (f32, f32,
f32, f32) {
  %cst = arith.constant 0.000000e+00 : f32
  %0 = arith.mulf %arg0, %arg0 : f32
  %1 = arith.mulf %arg1, %arg1 : f32
  %2 = arith.addf %arg2, %cst : f32
  %3 = arith.addf %arg3, %cst : f32
  %4 = arith.mulf %3, %arg1 : f32
  %5 = arith.addf %4, %cst : f32
  %6 = arith.mulf %3, %arg1 : f32
  %7 = arith.addf %5, %6 : f32
  %8 = arith.mulf %2, %arg0 : f32
  %9 = arith.addf %8, %cst : f32
  %10 = arith.mulf %2, %arg0 : f32
  %11 = arith.addf %9, %10 : f32
  return %0, %1, %11, %7 : f32, f32, f32, f32
}
```

compute primals

compute derivatives

Reverse Mode Activity Canonicalization

- Depending on the program context, we can modify the activity assignment to **`activity`** and **`ret_activity`**.
- ***Idea***: *Avoid unnecessary gradient computations, **before codegen***
- Before canonicalization, we **check variable uses** and **derivative values** (e.g. `dval = 0.0f`) to promote activity

Activity	Primal	Derivative
`active`	✅	✅
`activenoneed`	❌	✅
`const`	✅	❌
`constnoneed`	❌	❌

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

only need **o2** and **dx**

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

eliminate **dy**

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o1,%o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_active, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

eliminate **dy**

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o1,%o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_active, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

don't return **o1**

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o1,%o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_active, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_activenoneed, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

Reverse Mode Activity Canonicalization - Example

```
func @square(%x: f32, %y: f32) -> (f32, f32) {
  %o1 = arith.mulf %x, %x : f32
  %o2 = arith.mulf %y, %y : f32
  return %o1, %o2 : f32, f32
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
 %o1,%o2,%dx,%dy = enzyme.autodiff @square(%x,%y,%do1,%do2)
  { activity = [enzyme_active,enzyme_active],
    ret_activity = [enzyme_active,enzyme_active]
  }
  return %o2,%dx : (f32,f32)
}
```

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o1,%o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_active, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

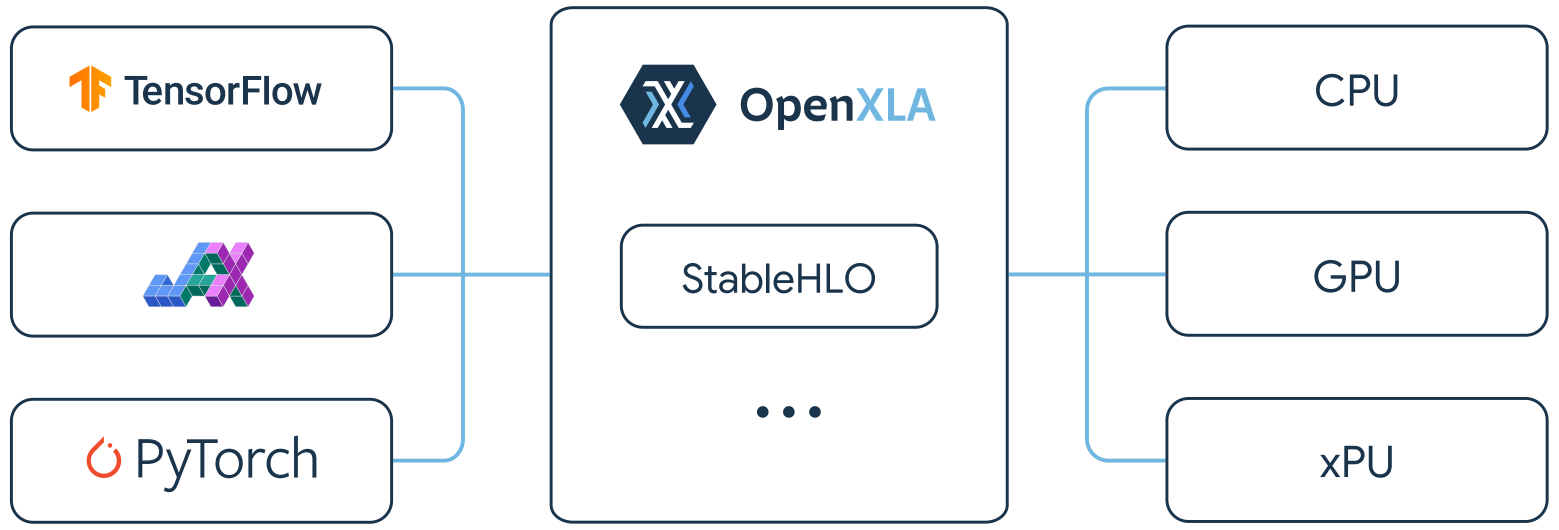
don't return **o1**

Activity	Primal	Derivative
active	✅	✅
activenoneed	❌	✅
const	✅	❌
constnoneed	❌	❌

```
func @grad_square(%x:f32, %y:f32, %do1: f32, %do2: f32) {
  %o2,%dx = enzyme.autodiff @square(%x,%y,%do1)
  { activity = [enzyme_active, enzyme_const],
    ret_activity = [enzyme_activenoneed, enzyme_const]
  }
  return %o2,%dx : (f32,f32)
}
```

Summary

- Enzyme approach (AD + compiler)
- EnzymeMLIR dialect
- Tensor Algebra Optimization
- Higher Order Derivatives
- Return Activity Canonicalization




```
func @grad_set(%X: memref<?xf32>, %v: f32, %dout: f32) {
 %out, %dv = enzyme.autodiff @set(%X,%v,%dout) {
    activity = [enzyme_const, enzyme_active]
    ret_activity = [enzyme_active]
 } : (f32, f32)
 return
}
```

Backup slides

+

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

$O\left(n^2\right)$

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

Optimization & Automatic Differentiation

$O\left(n^2\right)$

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

$O\left(n^2\right)$

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

AD

$O\left(n^2\right)$

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

Optimization & Automatic Differentiation

$O\left(n^2\right)$

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

$O\left(n^2\right)$

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

AD

$O\left(n^2\right)$

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

Optimize

$O\left(n^2\right)$

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

Optimization & Automatic Differentiation

Differentiating after optimization can create ***asymptotically faster*** gradients!

$O\left(n^2\right)$

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

$O\left(n^2\right)$

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

AD

$O\left(n^2\right)$

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

Optimize

$O\left(n^2\right)$

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