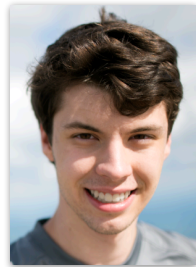
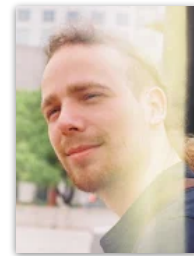




# Post-Optimization Automatic Differentiation by Synthesizing LLVM



**William S. Moses**



Valentin Churavy

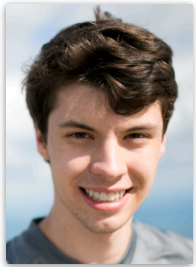


wmoses@mit.edu

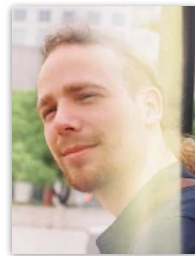
Differentiable Programming Workshop

April 7, 2021





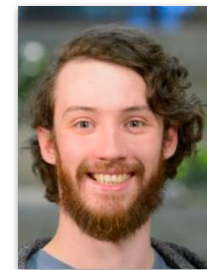
William S. Moses



Valentin Churavy



Ludger Paehler



Johannes Doerfert



Jan Hückelheim



Sri Hari Krishna  
Narayanan



Michel Schanen



Paul Hovland

# Differentiation Is Key To Machine Learning And Science

---

- Computing derivatives is key to many algorithms
  - Machine learning (back-propagation, Bayesian inference, uncertainty quantification)
  - Scientific computing (modeling, simulation)
- When working with large codebases or dynamically-generated programs, manually writing derivative functions becomes intractable
- Community has developed tools to create derivatives automatically



# Existing AD Approaches

---

- 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
- Operator overloading (Adept, JAX)
  - Provide 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





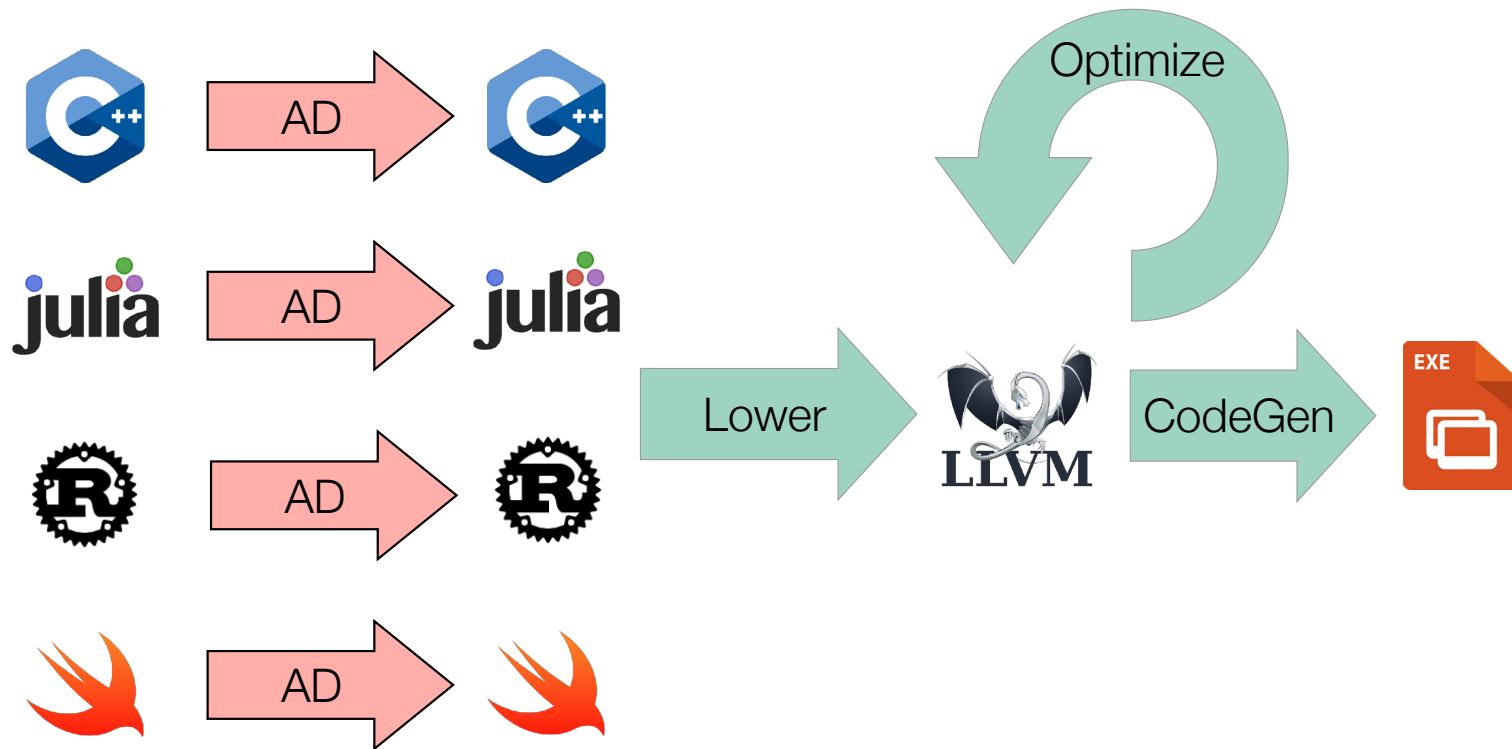
# Existing AD Approaches

---

- 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
  - Difficult to use with external libraries



# Existing Automatic Differentiation Pipelines



## Case Study: Vector Normalization

---

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

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

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

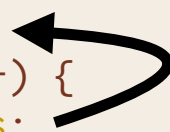


## Case Study: Vector Normalization

---

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

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



# Optimization & Automatic Differentiation

---

$O(n^2)$

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

Optimize

$O(n)$

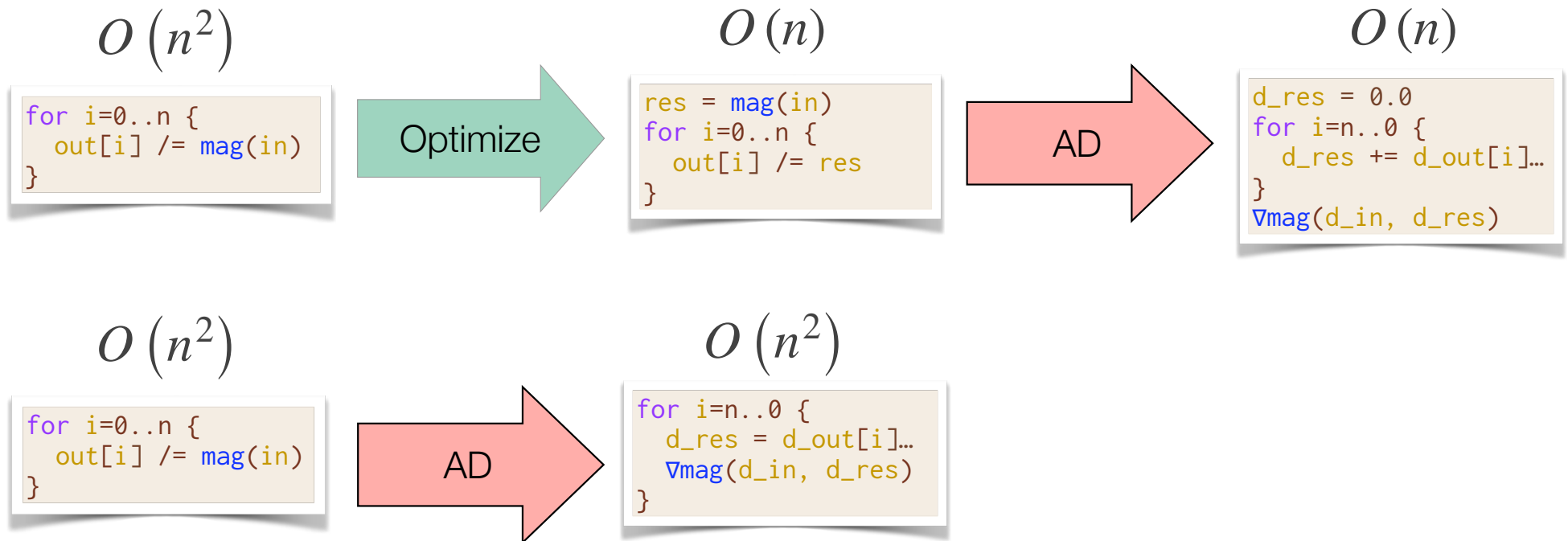
```
res = mag(in)  
for i=0..n {  
  out[i] /= res  
}
```

AD

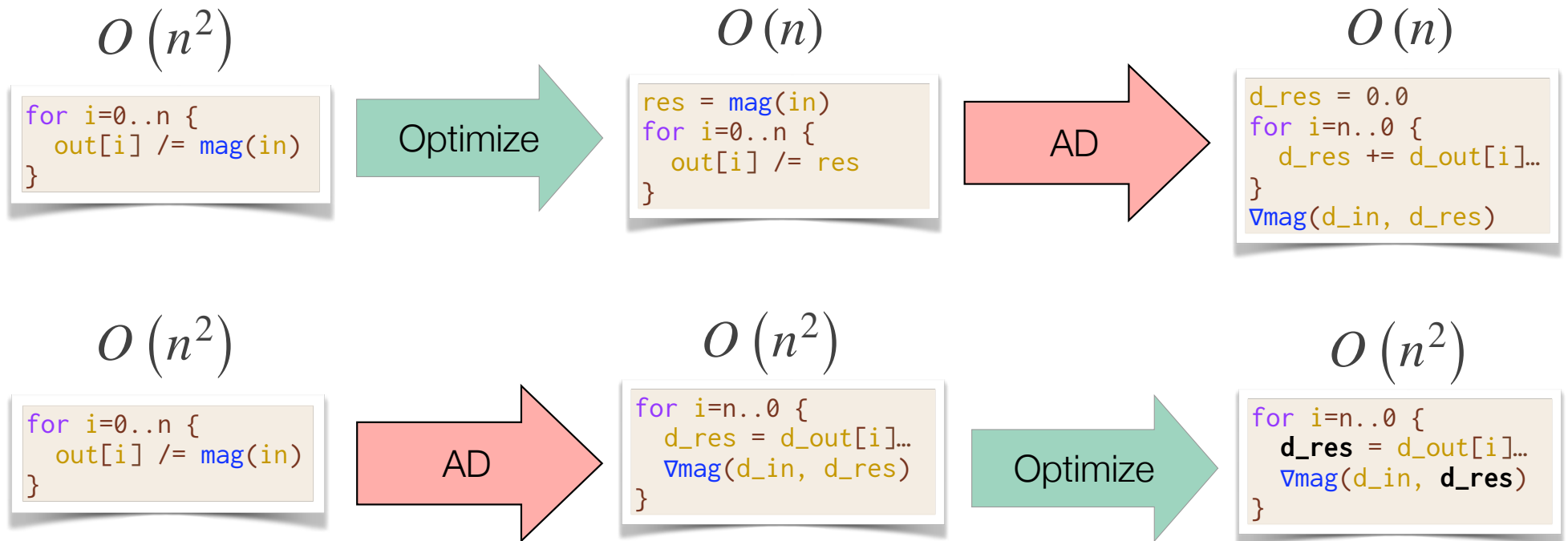
$O(n)$

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

# Optimization & Automatic Differentiation

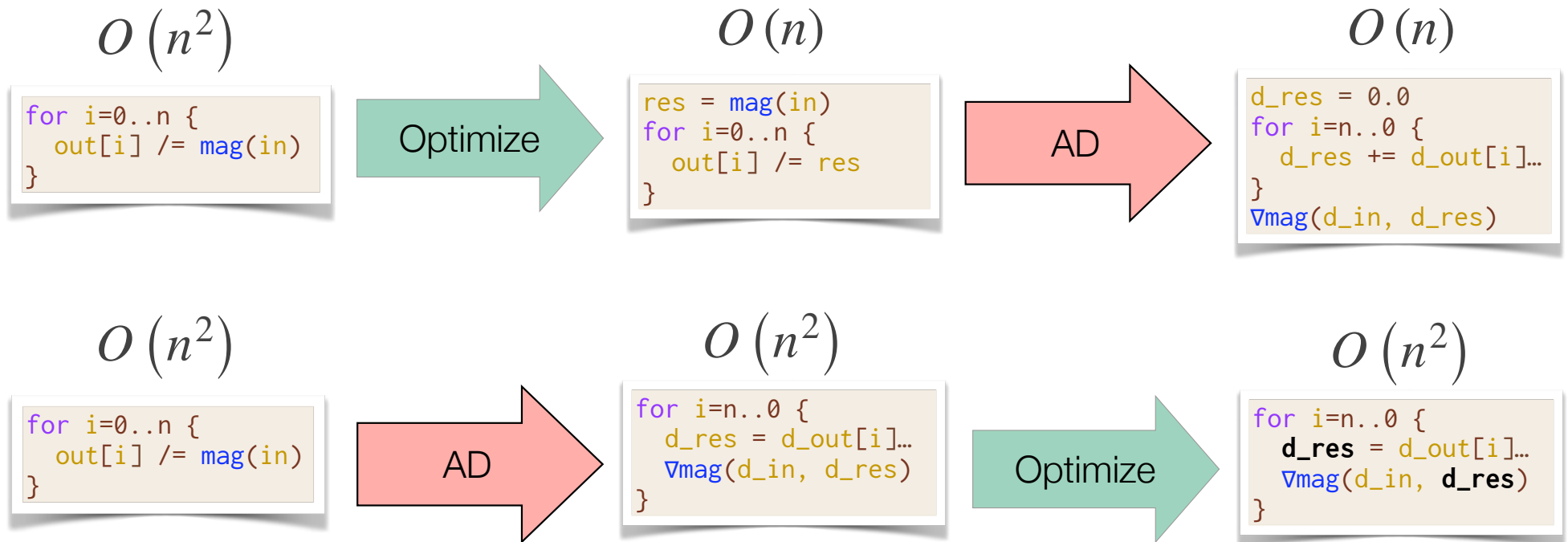


# Optimization & Automatic Differentiation



# Optimization & Automatic Differentiation

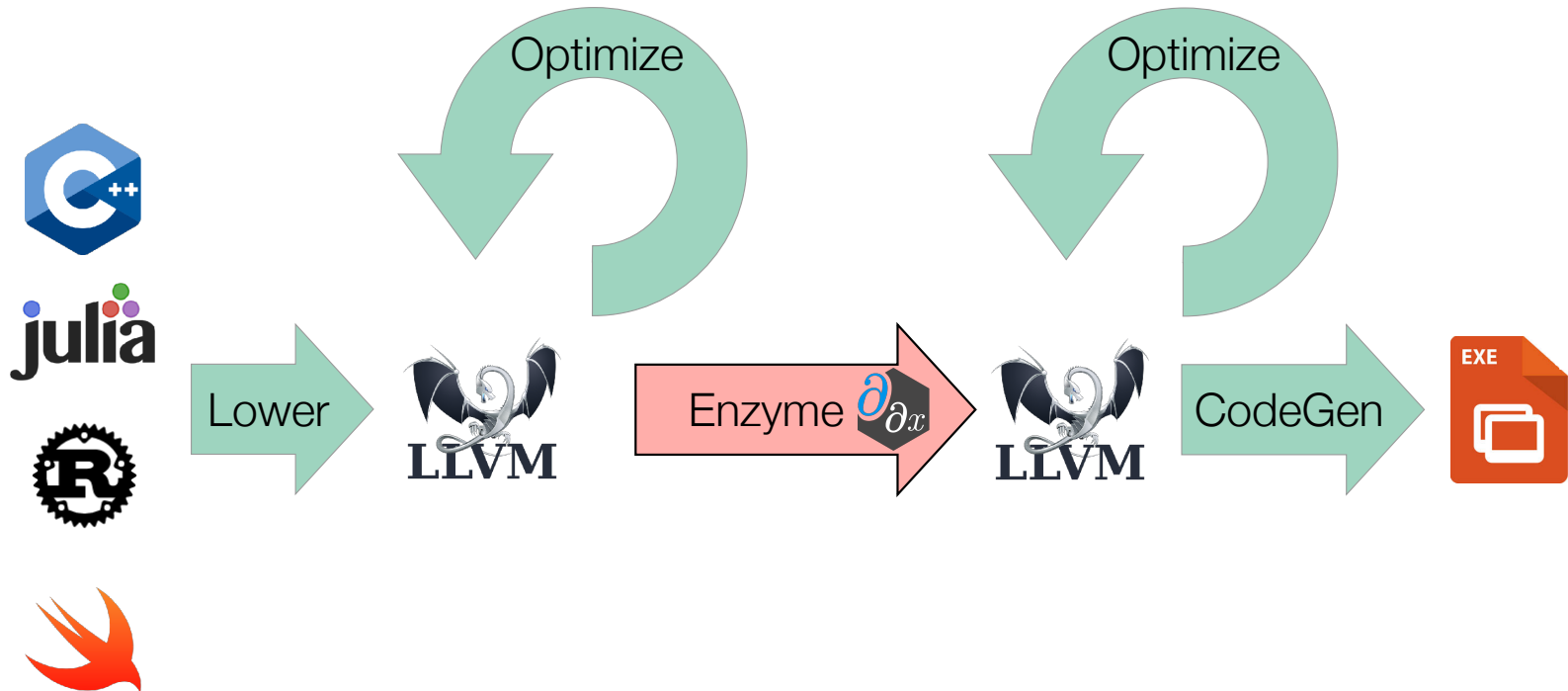
Differentiating after optimization can create *asymptotically faster* gradients!





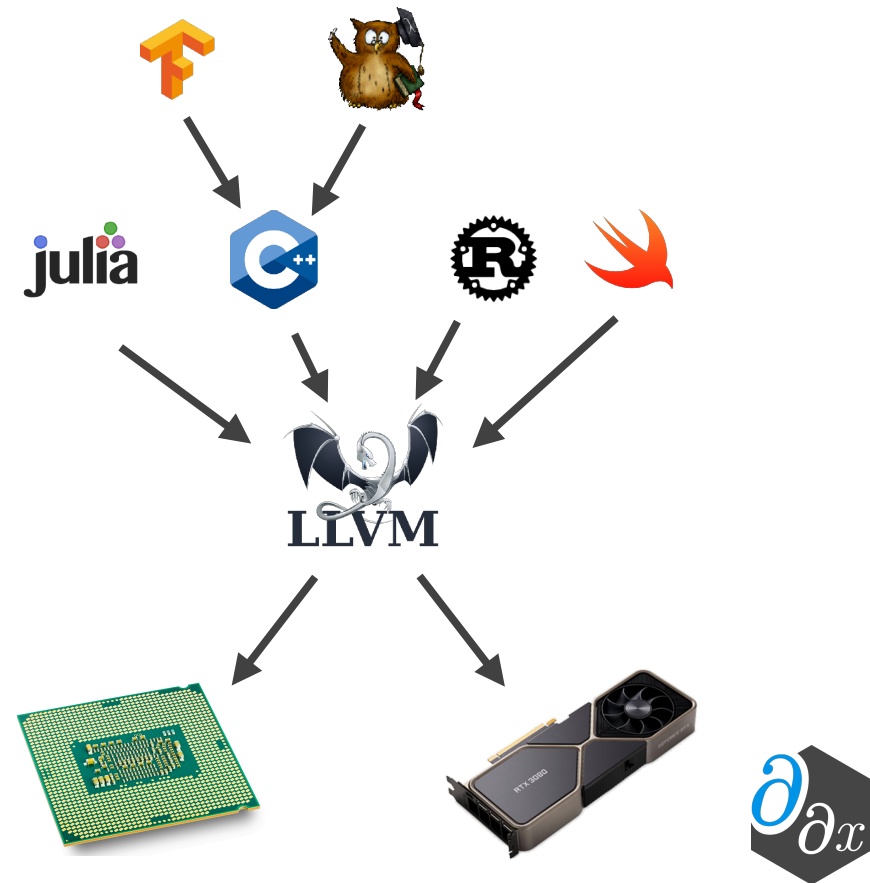
# Enzyme Approach

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, etc)
- Well-defined semantics
- Large collection of optimizations and analyses



# Case Study: ReLU3

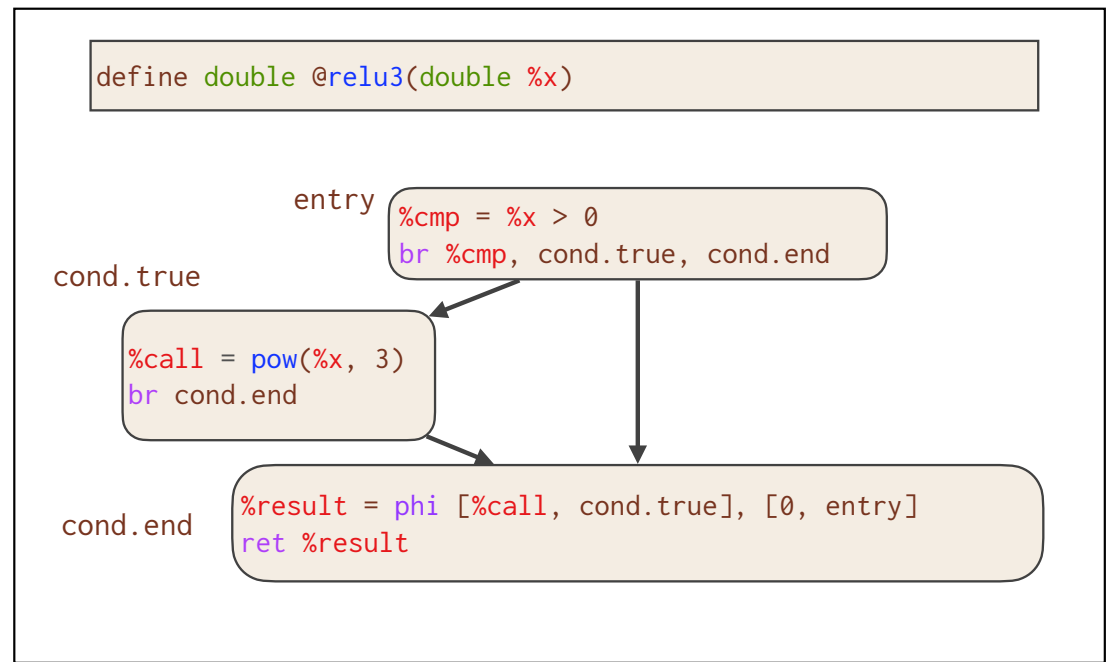
## C Source

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

## Enzyme Usage

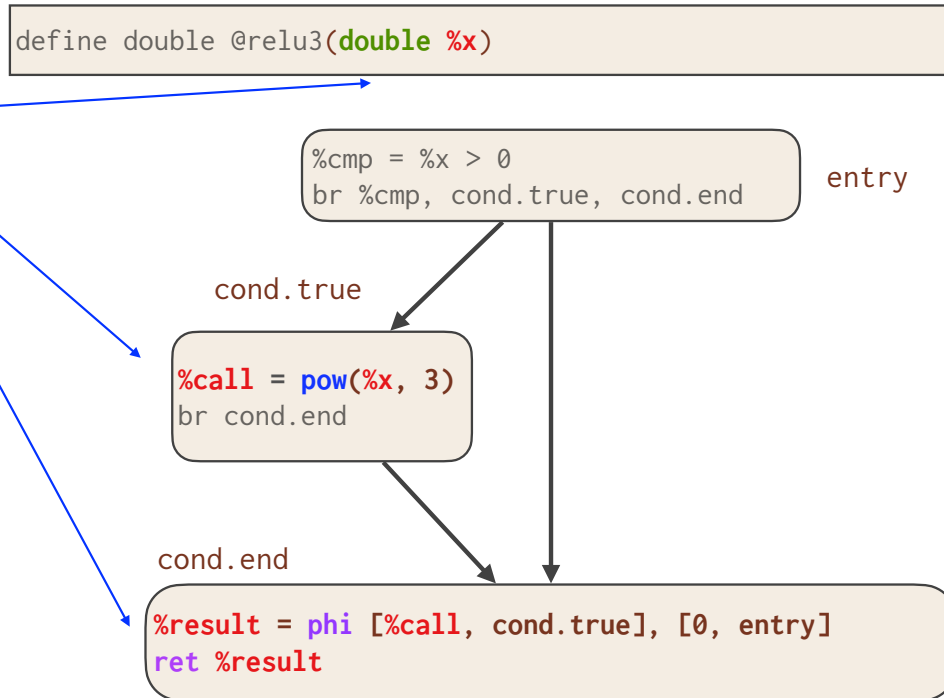
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

## LLVM

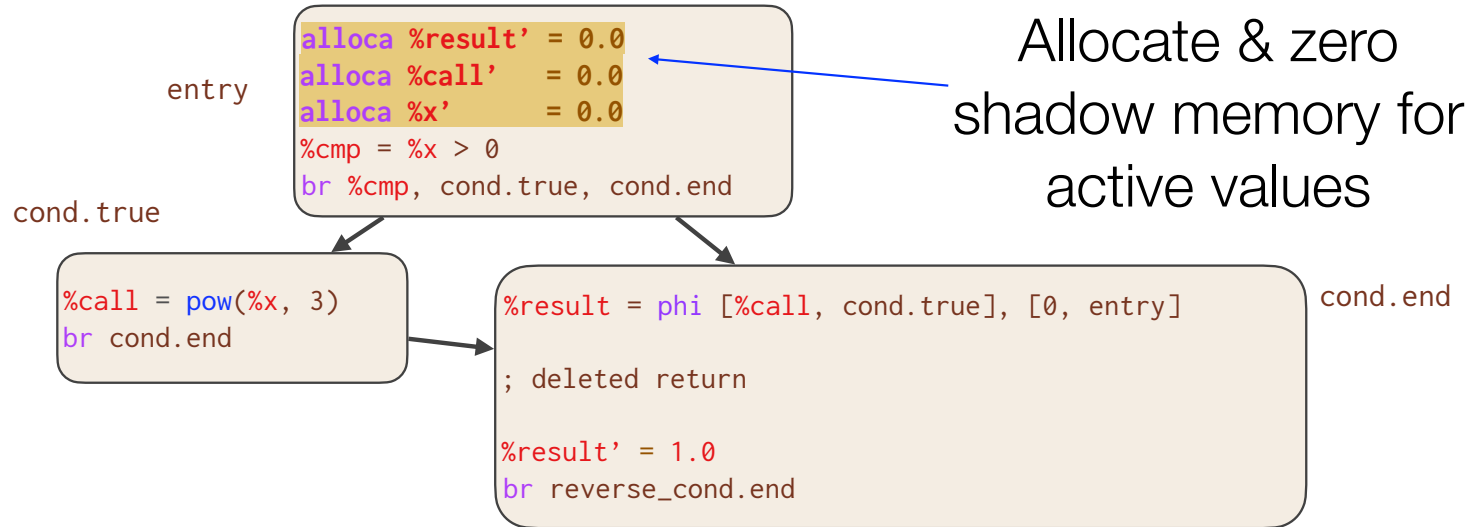


# Case Study: ReLU3

Active Instructions

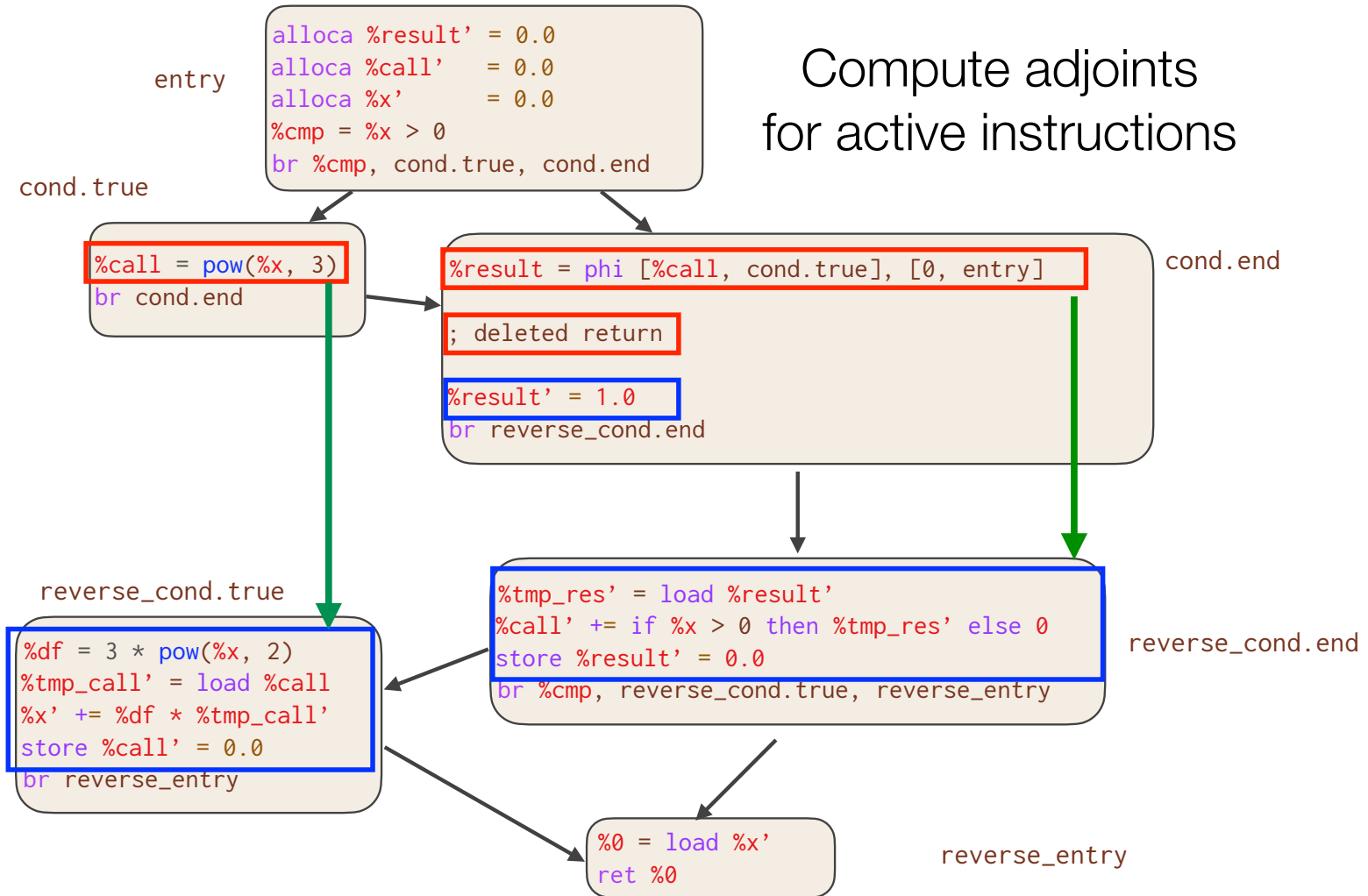


```
define double @diffe_relu3(double %x, double %differet)
```



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

Compute adjoints  
for active instructions



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

entry

```
alloca %result' = 0.0  
alloca %call' = 0.0  
alloca %x' = 0.0  
%cmp = %x > 0  
br %cmp, cond.true, cond.end
```

Compute adjoints  
for active instructions

cond.true

```
%call = pow(%x, 3)  
br cond.end
```

```
%result = phi [%call, cond.true], [0, entry]  
; deleted return  
%result' = 1.0  
br reverse_cond.end
```

cond.end

reverse\_cond.true

```
%df = 3 * pow(%x, 2)  
%tmp_call' = load %call  
%x' += %df * %tmp_call'  
store %call' = 0.0  
br reverse_entry
```

```
%tmp_res' = load %result'  
%call' += if %x > 0 then %tmp_res' else 0  
store %result' = 0.0  
br %cmp, reverse_cond.true, reverse_entry
```

reverse\_cond.end

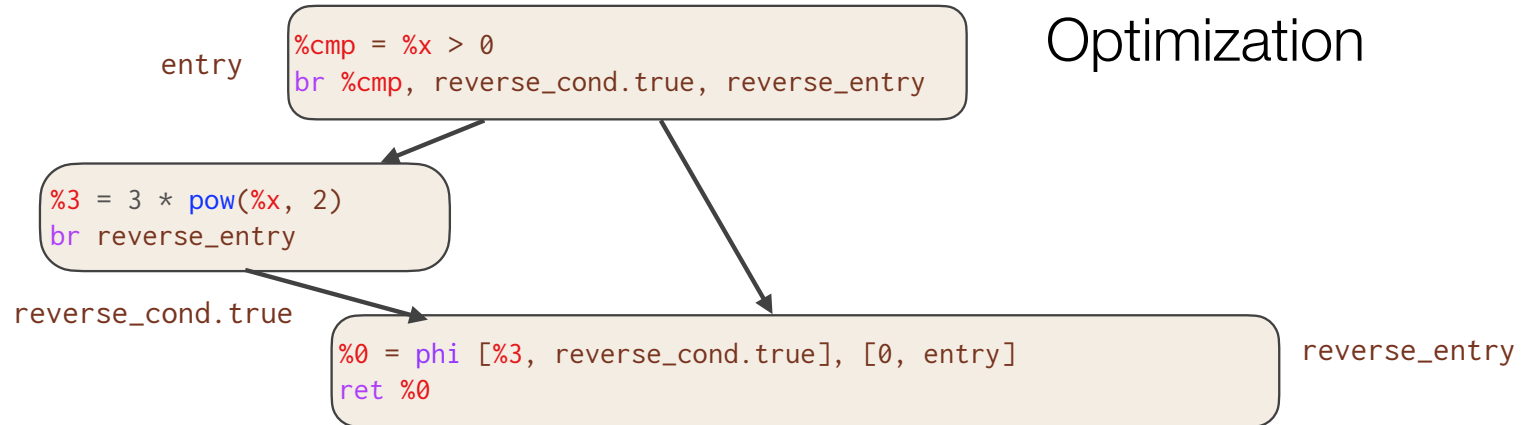
```
%0 = load %x'  
ret %0
```

reverse\_entry



```
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;  
}
```





# Challenges of Low-Level AD

- Low-level code lacks information necessary to compute adjoints

```
void f(void* dst, void* src) {  
    memcpy(dst, src, 8);  
}
```

```
void grad_f(double* dst, double* dst',  
            double* src, double* src') {  
    // Forward Pass  
    memcpy(dst, src, 8);  
  
    // Reverse Pass  
    src'[0] += dst'[0];  
    dst'[0] = 0;  
}
```

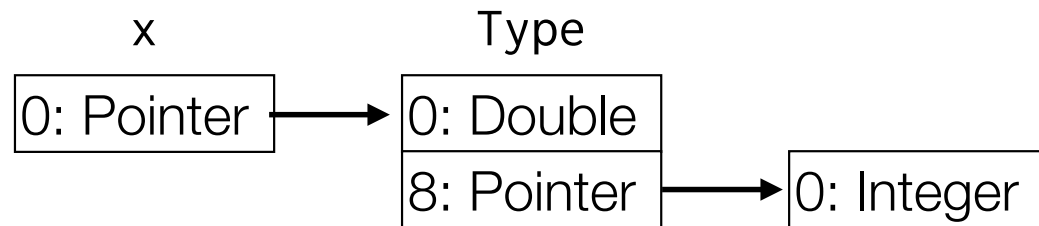
```
void grad_f(float* dst, float* dst',  
            float* src, float* src') {  
    // Forward Pass  
    memcpy(dst, src, 8);  
  
    // Reverse Pass  
    src'[0] += dst'[0];  
    dst'[0] = 0;  
    src'[1] += dst'[1];  
    dst'[1] = 0;  
}
```



# Challenges of Low-Level AD

- New interprocedural dataflow analysis that detects the underlying type of data
- Each value has a set of memory offsets : type
- Perform series of fixed-point updates through instructions

```
struct Type {  
    double;  
    int*;  
}  
  
x = Type*;
```



$\text{types}(x) = \{[0]:\text{Pointer}, [0,0]:\text{Double}, [0,8]:\text{Pointer}, [0,8,0]:\text{Integer}\}$

# Custom Derivatives & Multisource

---

- One can specify custom forward/reverse passes of functions by attaching metadata

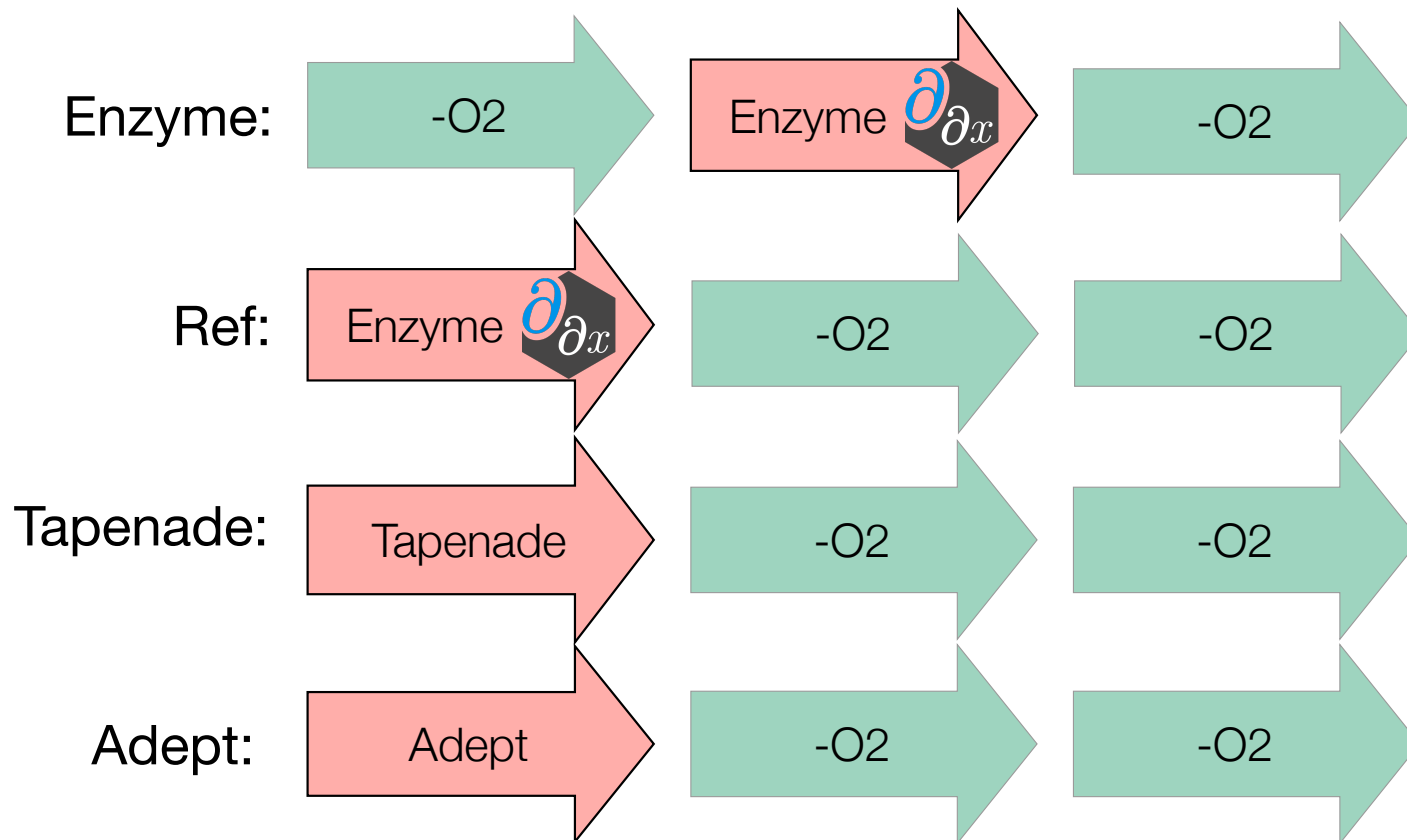
```
__attribute__((enzyme("augment", augment_func)))  
__attribute__((enzyme("gradient", gradient_func)))  
double func(double n);
```

- Enzyme leverages LLVM's link-time optimization (LTO) & "fat libraries" to ensure that LLVM bitcode is available for all potential differentiated functions before AD

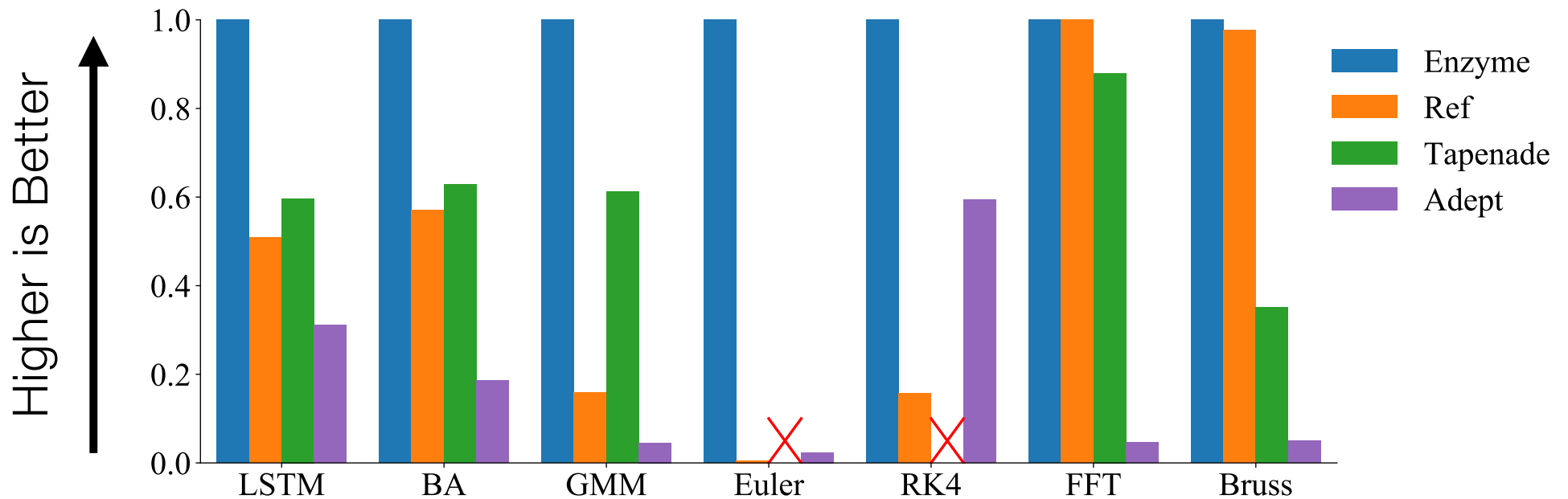


# Experimental Setup

- Collection of benchmarks from Microsoft's ADBench suite and of technical interest



## Speedup of Enzyme



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



# PyTorch-Enzyme & TensorFlow-Enzyme

```
import torch
from torch_enzyme import enzyme

# Create some initial tensor
inp = ...

# Apply foreign function to tensor
out = enzyme("test.c", "f").apply(inp)

# Derive gradient
out.backward()
print(inp.grad)
```

```
import tensorflow as tf
from tf_enzyme import enzyme

# Create some initial tensor
inp = tf.Variable(...)

# Use external C code as a regular TF op
out = enzyme(inp, filename="test.c",
             function="f")

# Results is a TF tensor
out = tf.sigmoid(out)
```

```
// Input tensor + size, and output tensor
void f(float* inp, size_t n, float* out);

// diffe_dupnoneed specifies not recomputing the output
void diffef(float* inp, float* d_inp, size_t n, float* d_out) {
    __enzyme_autodiff(f, diffe_dup, inp, d_inp, n, diffe_dupnoneed, (float*)0, d_out);
}
```





- Tool for performing reverse-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization
- State-of-the art performance with existing tools
- Differentiate GPU kernels (come to GTC talk for more info!)
- Open Source ([enzyme.mit.edu](http://enzyme.mit.edu) / [github.com/wsmoses/Enzyme](https://github.com/wsmoses/Enzyme))
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow

# Acknowledgements

---

- Thanks to James Bradbury, Alex Chernyakhovsky, Hal Finkel, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Miguel Young de la Sota, and Alex Zinenko
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323.
- Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443.
- This research was supported in part by LANL grant 531711. Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.
- The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.







- Tool for performing reverse-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization
- State-of-the art performance with existing tools
- Differentiate GPU kernels (come to GTC talk for more info!)
- Open Source ([enzyme.mit.edu](http://enzyme.mit.edu) / [github.com/wsmoses/Enzyme](https://github.com/wsmoses/Enzyme))
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow

**END**

---

