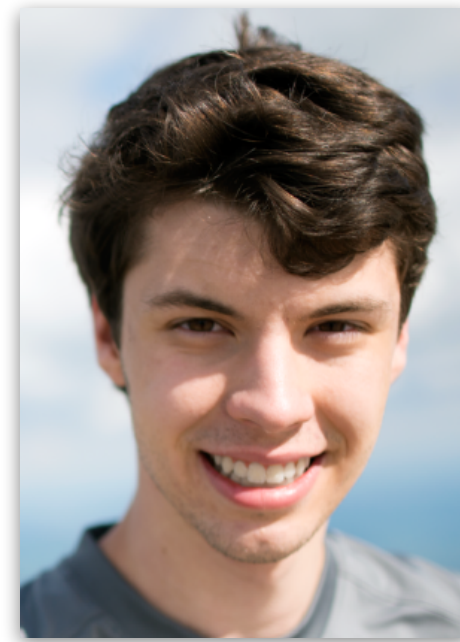




EnzymeMLIR: High-Performance Automatic Differentiation of Tensor Code



William S. Moses

wsmoses@illinois.edu

ICCOPT

Jul 23, 2025



UNIVERSITY OF
ILLINOIS
URBANA-CHAMPAIGN



Outline

- Compiler-Based Differentiation (Enzyme-LLVM)



- What is MLIR?



- Compiler-Based Differentiation With High-Level Operations (EnzymeMLIR)



- Some Fun Results

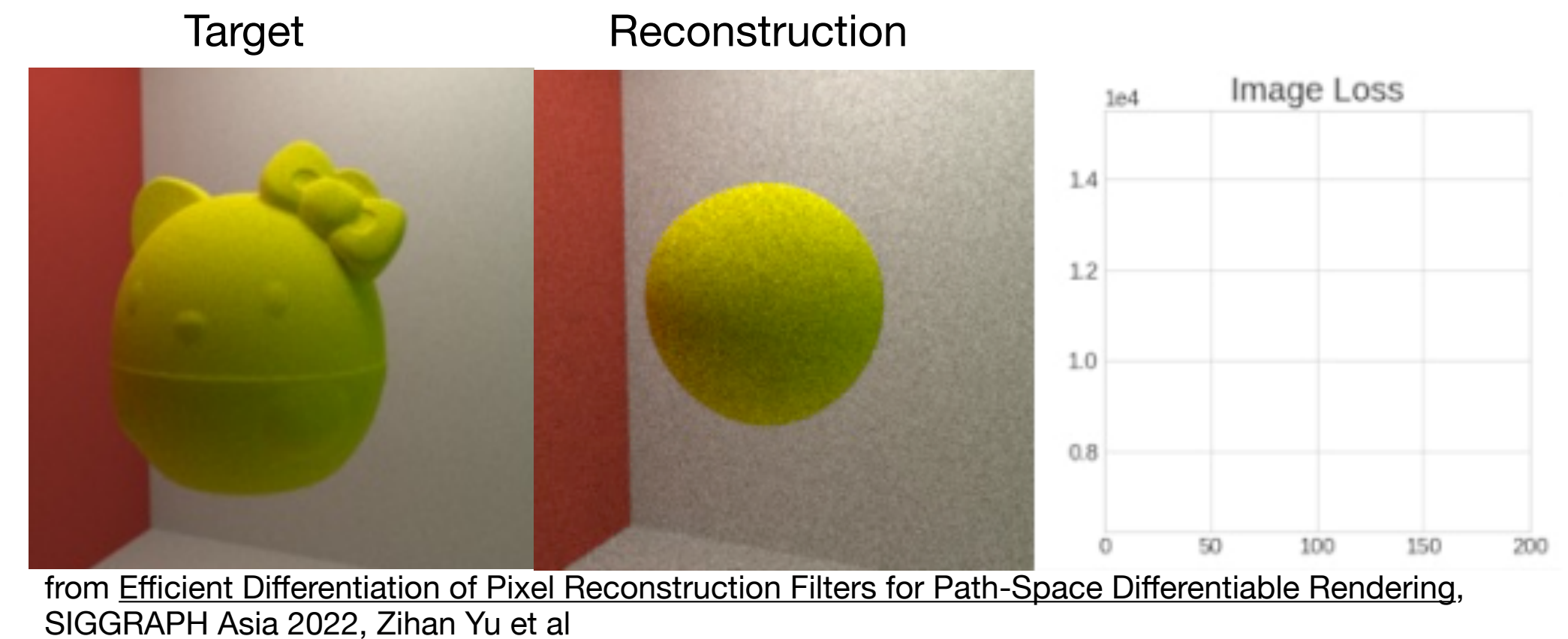




Differentiation: Connecting Science and AI

Derivatives are key to science + ML

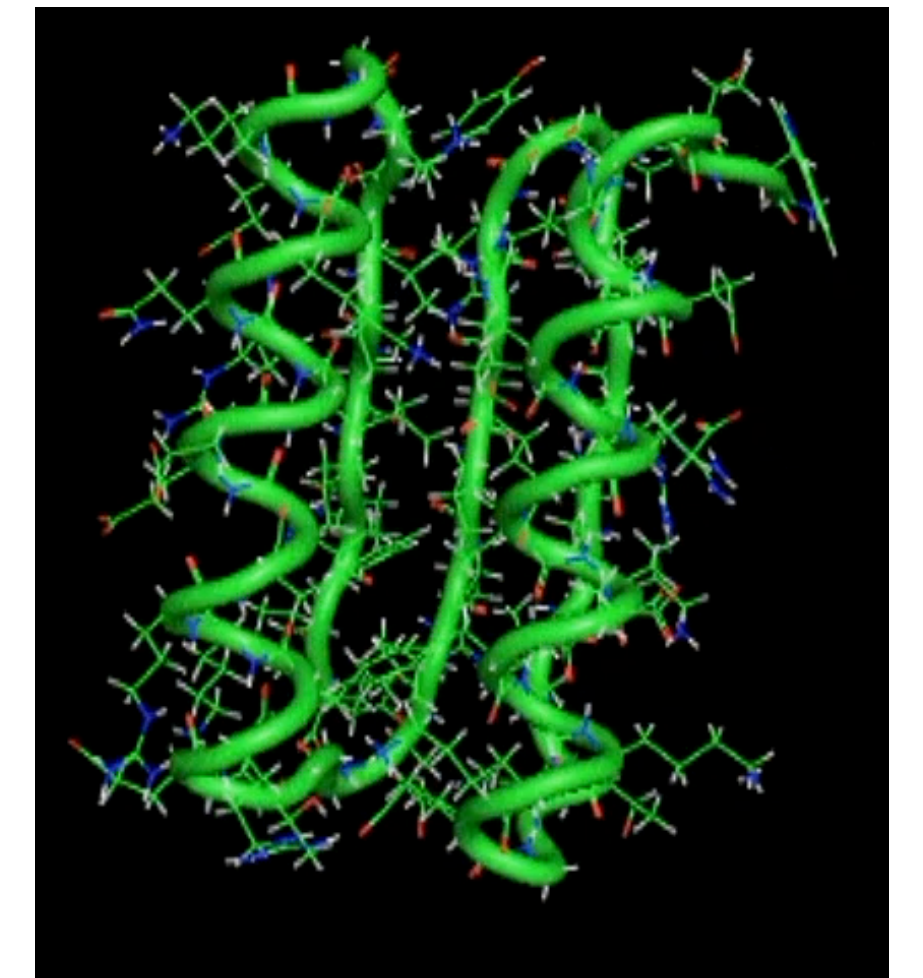
- Scientific Computing: UQ, Differential Equation, Error Analysis
- Machine Learning: Back-Propagation, Bayesian Inference



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



from Center for the Exascale Simulation of Materials in Extreme Environments

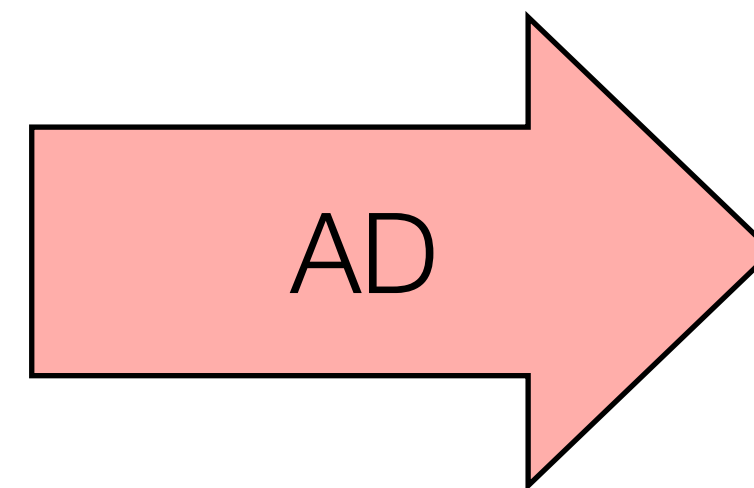


from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

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)
```

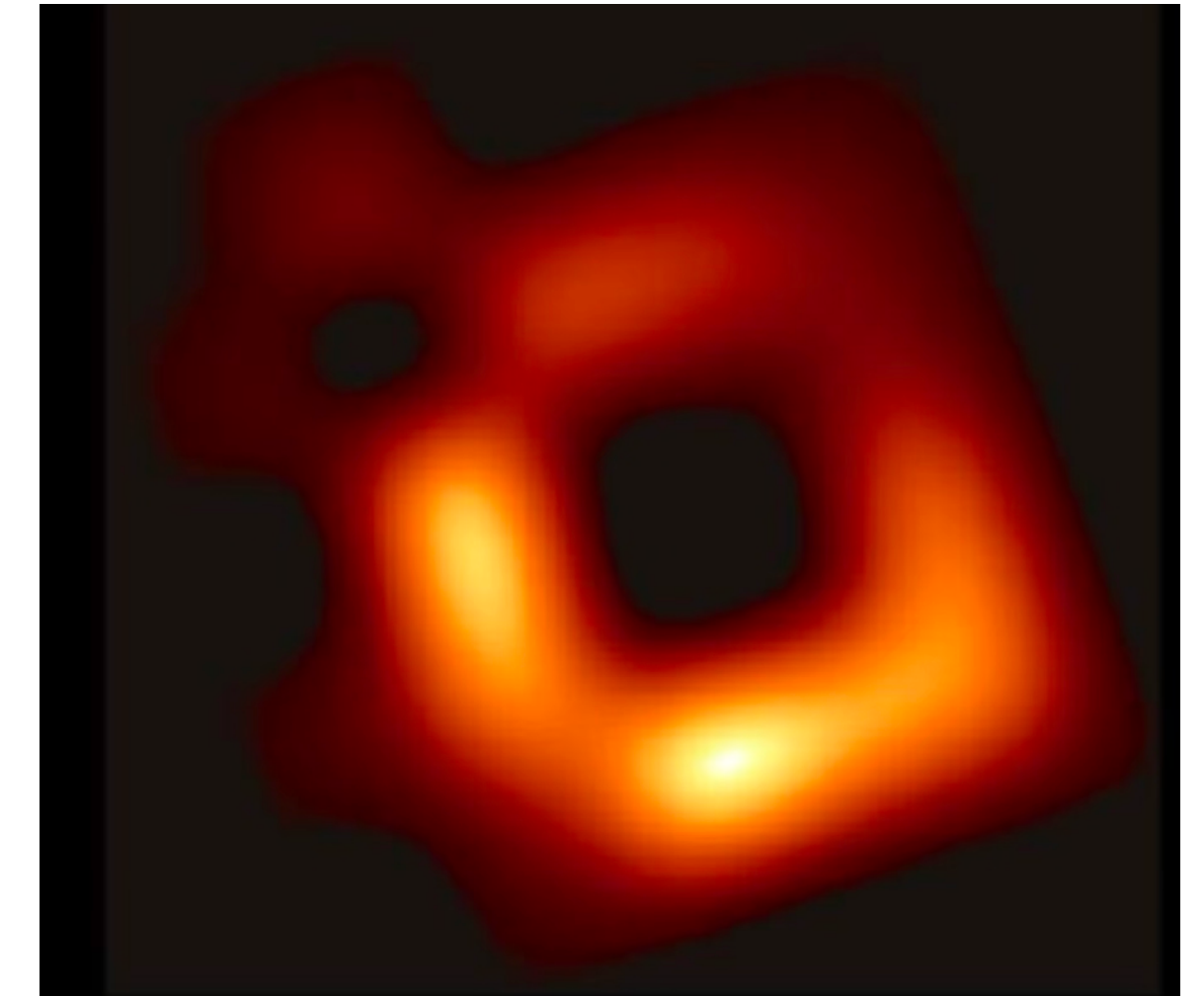
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

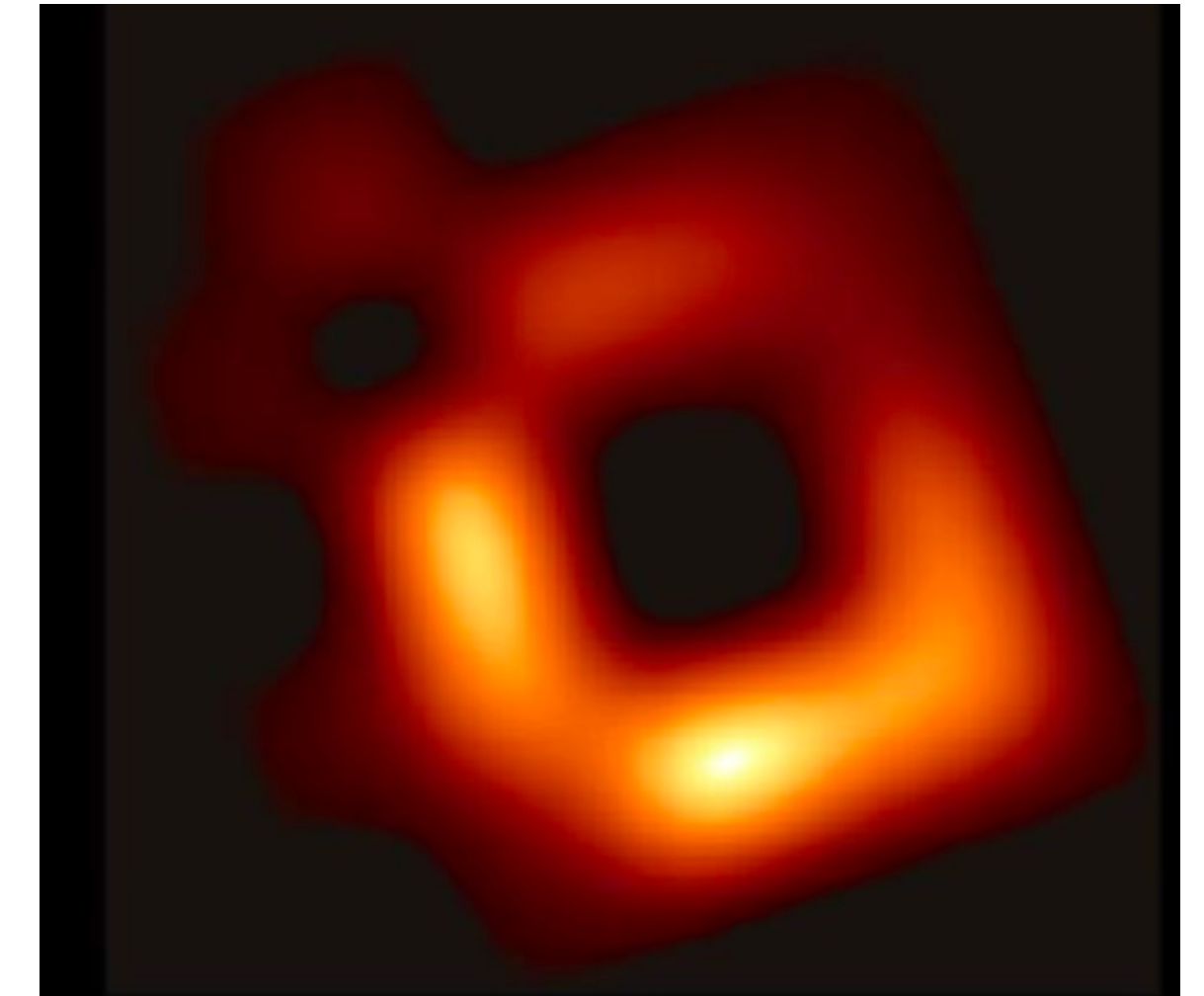
Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



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



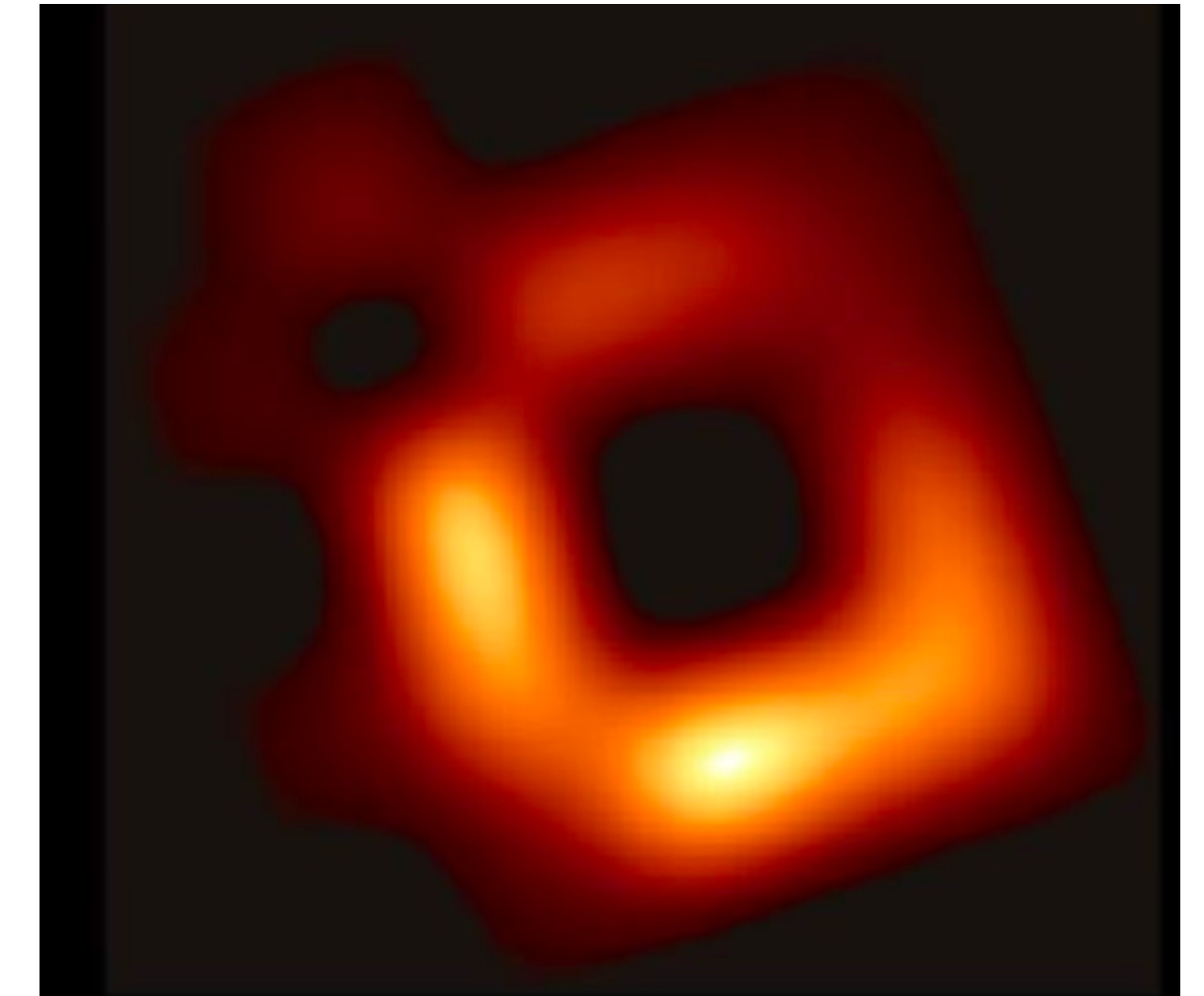
With Enzyme differentiation:
1 hour on 1 thread



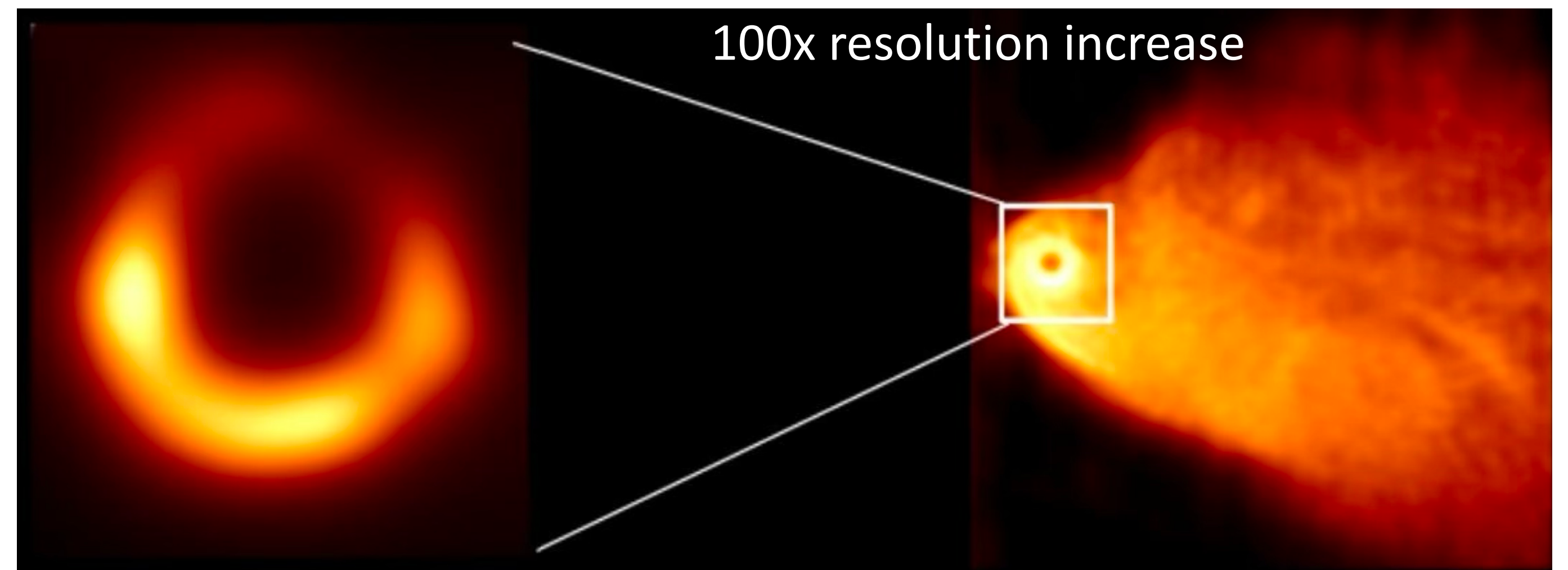
Differentiation is Expensive

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;
}
```

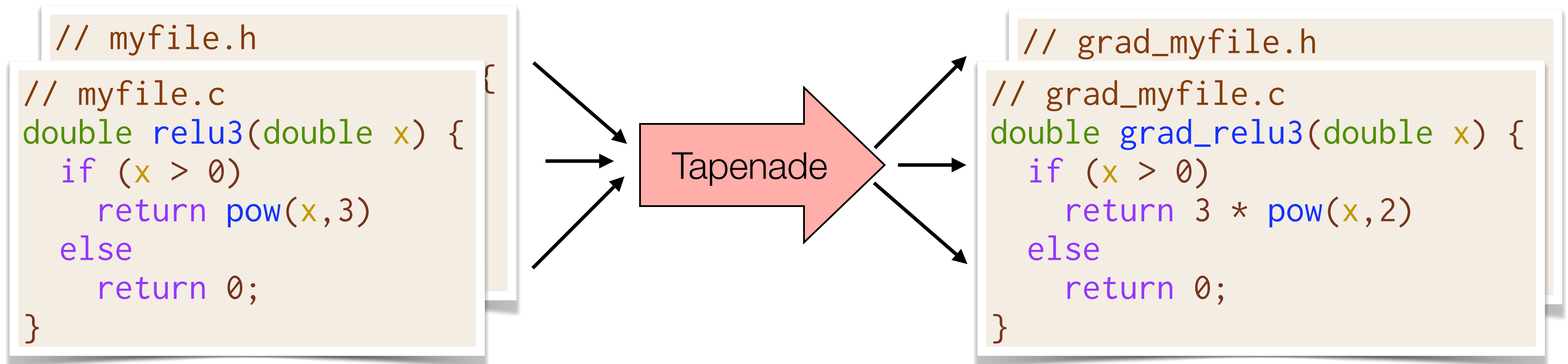
```
adept::Stack stack;
adept::adouble inp = 3.14;

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

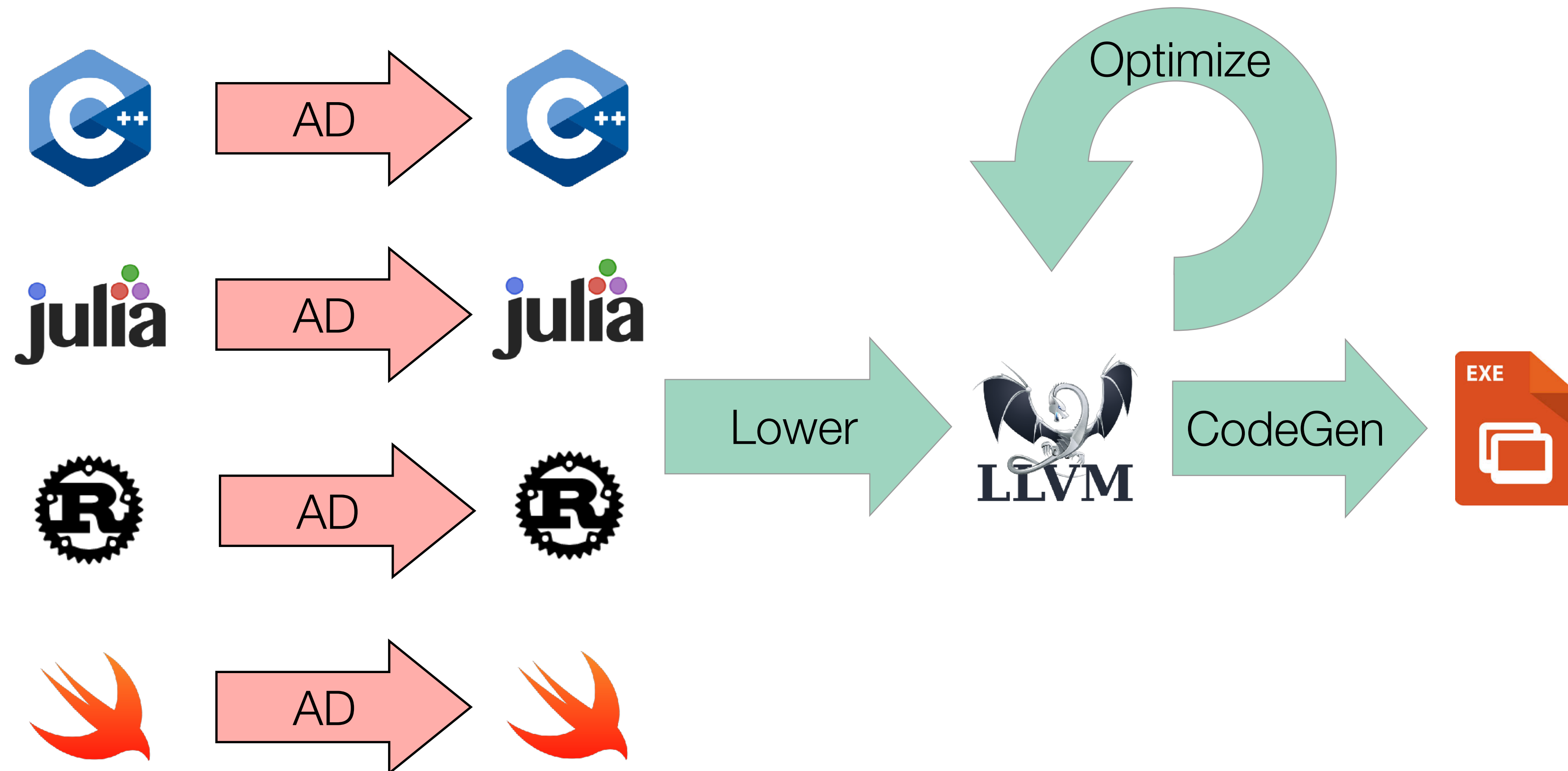
// Interpret all stack instructions
double res = inp.get_gradient(3.14);
```

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

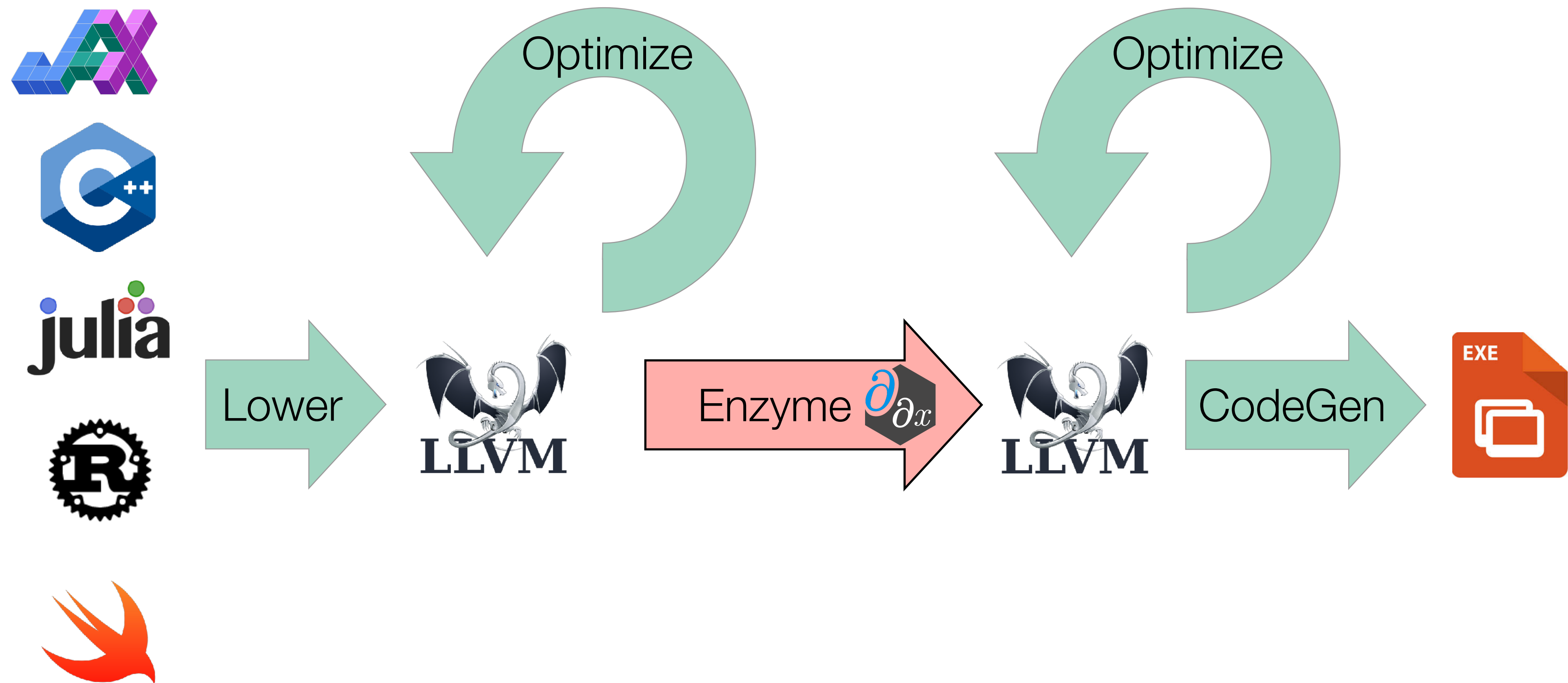


Existing Automatic Differentiation Pipelines



Enzyme Approach

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



Case Study: Vector Normalization

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

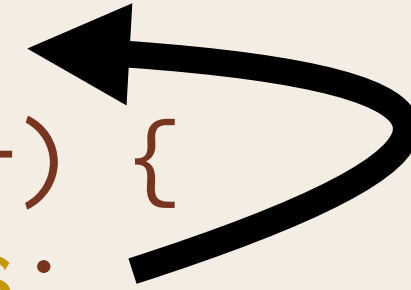
//Compute norm in O(n^2)
void norm(double[] out, double[] in) {

    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

$O(n^2)$

```
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)
```

$O(n^2)$

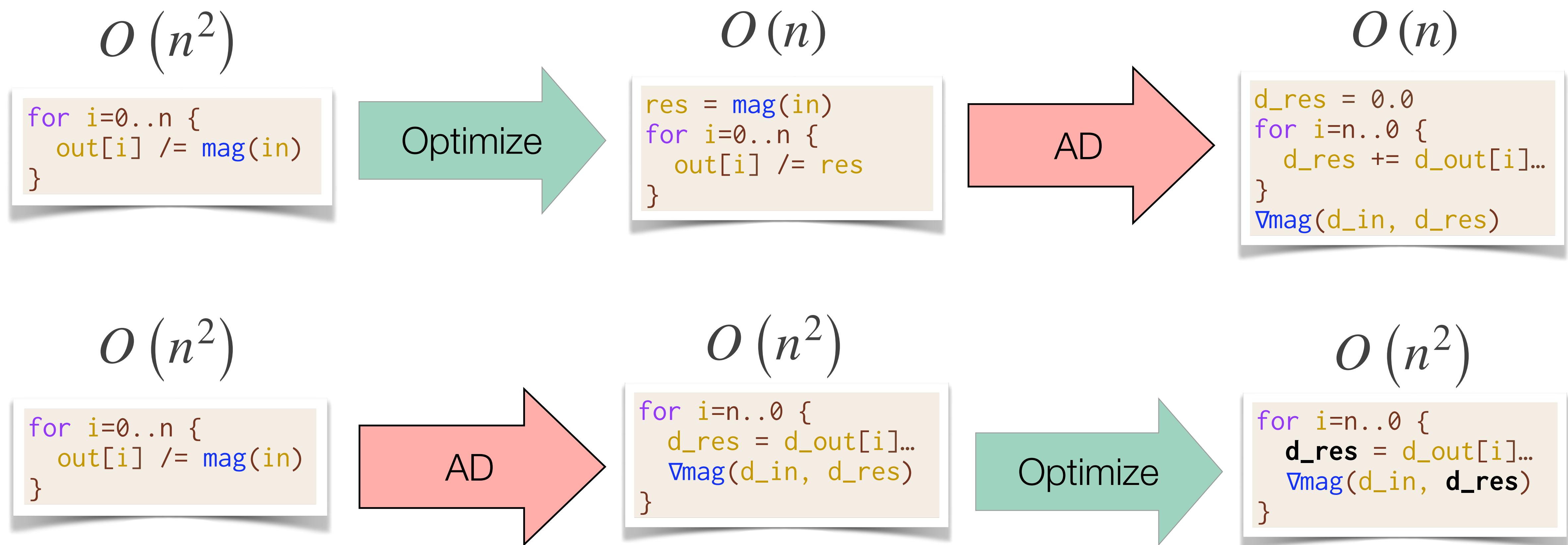
```
for i=0..n {  
  out[i] /= mag(in)  
}
```

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!

$O(n^2)$

```
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

$O(n^2)$

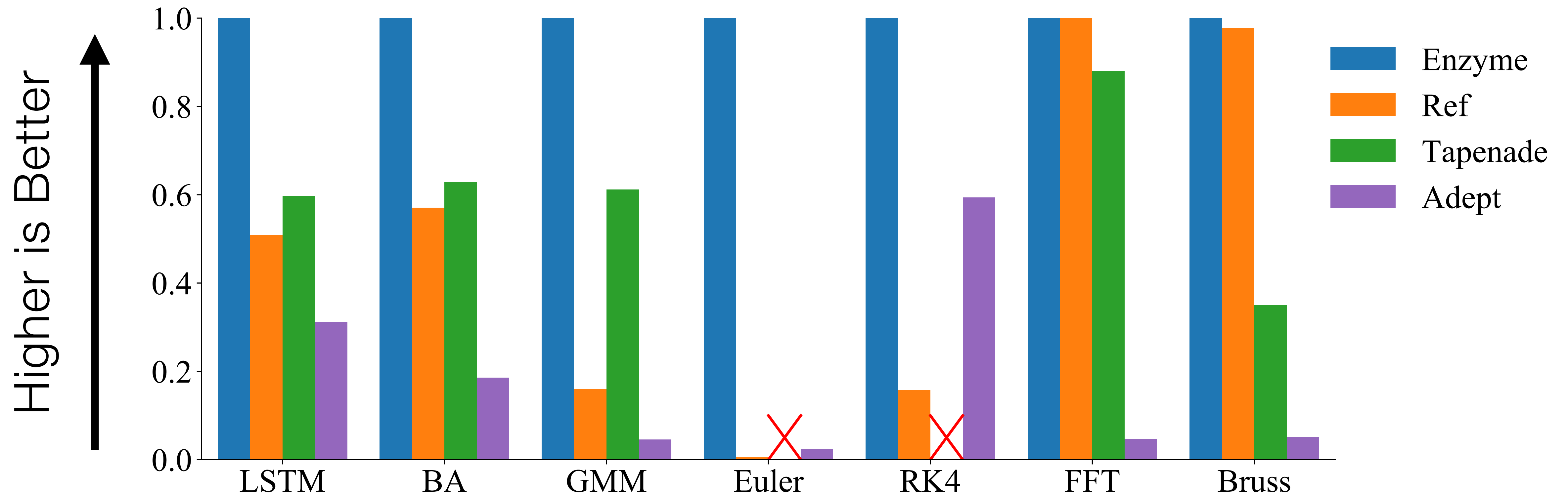
```
for i=n..0 {  
  d_res = d_out[i]...  
  ∇mag(d_in, d_res)  
}
```

Optimize

$O(n^2)$

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

Enzyme CPU Speedups [NeurIPS'20]



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

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)

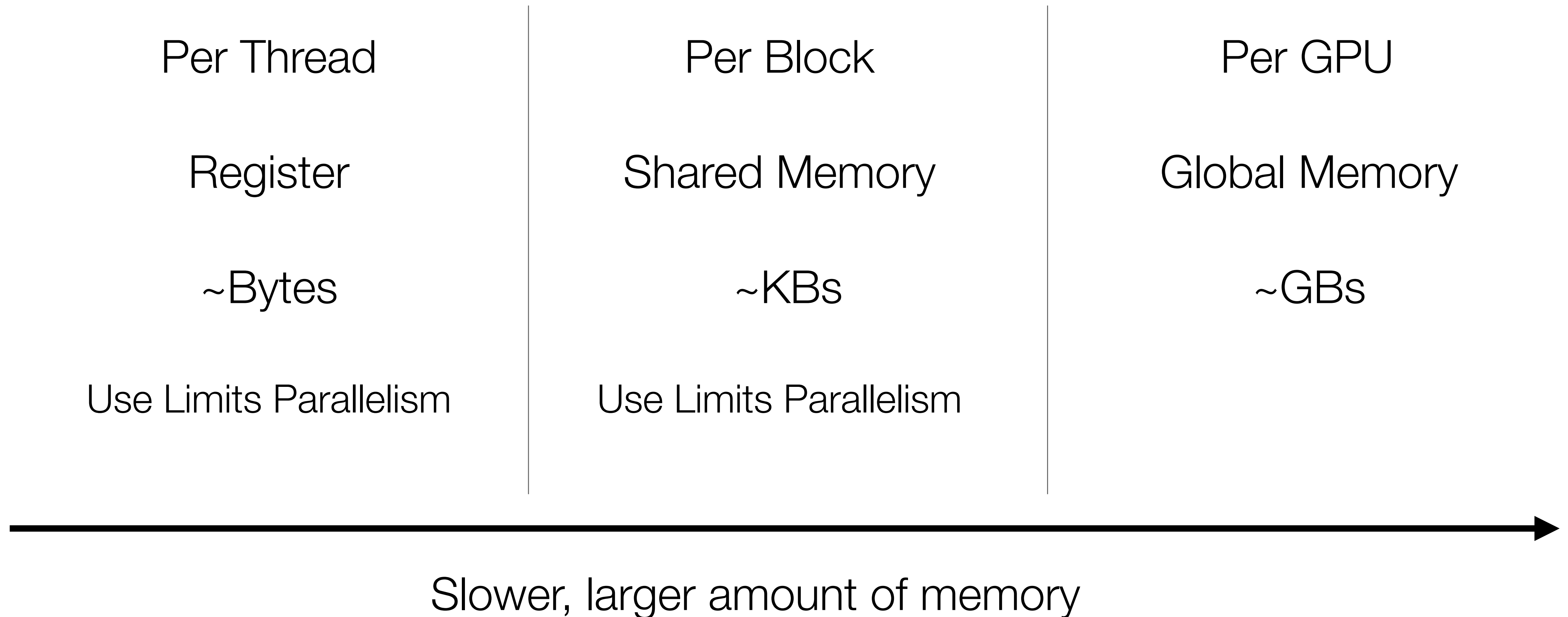
```
void set(double* ar, double val) {  
    parallel_for(int i=0; i<10; i++)  
        ar[i] = val;  
}
```

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; }  
}
```

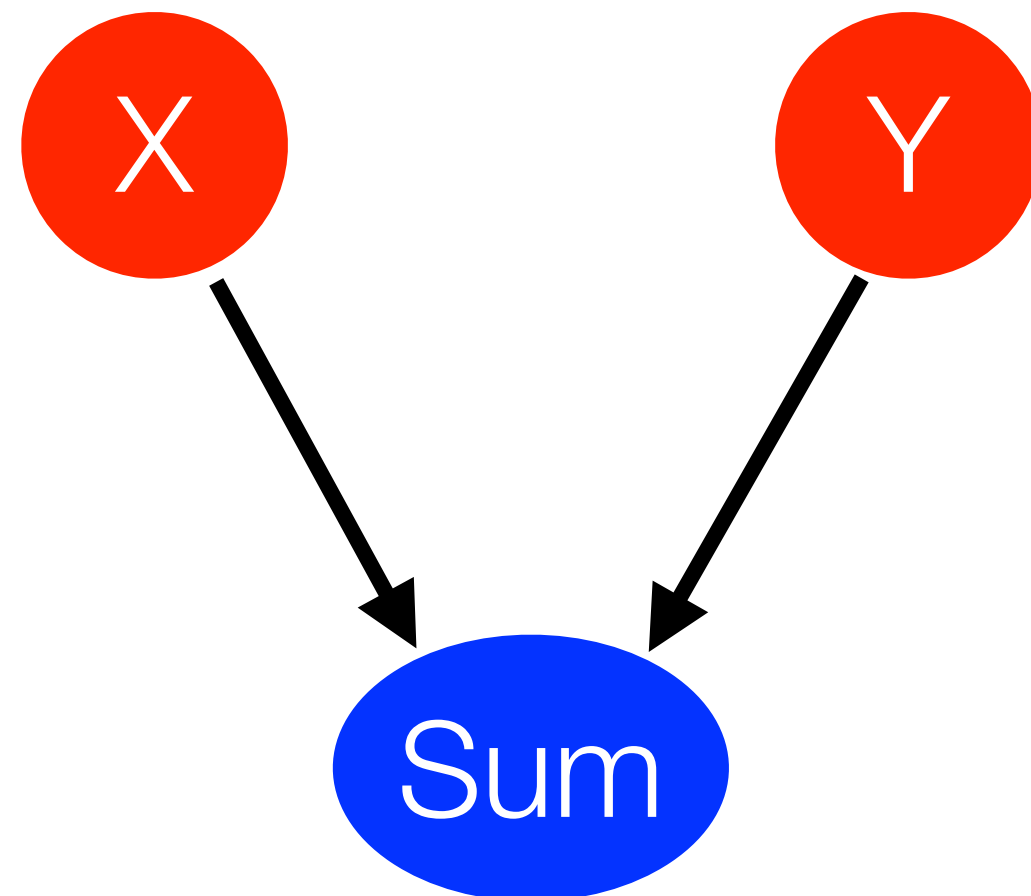
Slower



Cache Reduction Example

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

Overwritten:

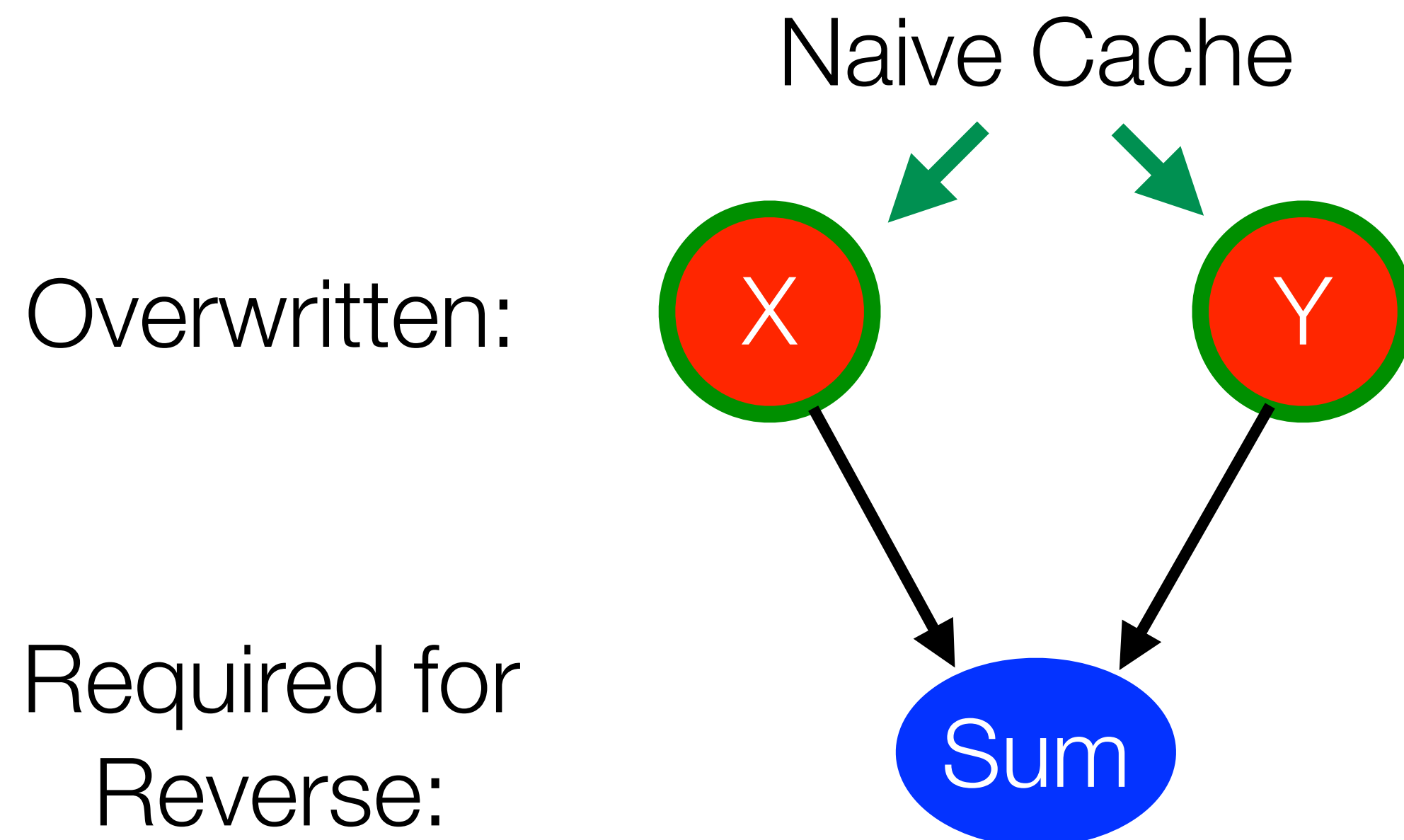


Required for
Reverse:

```
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);

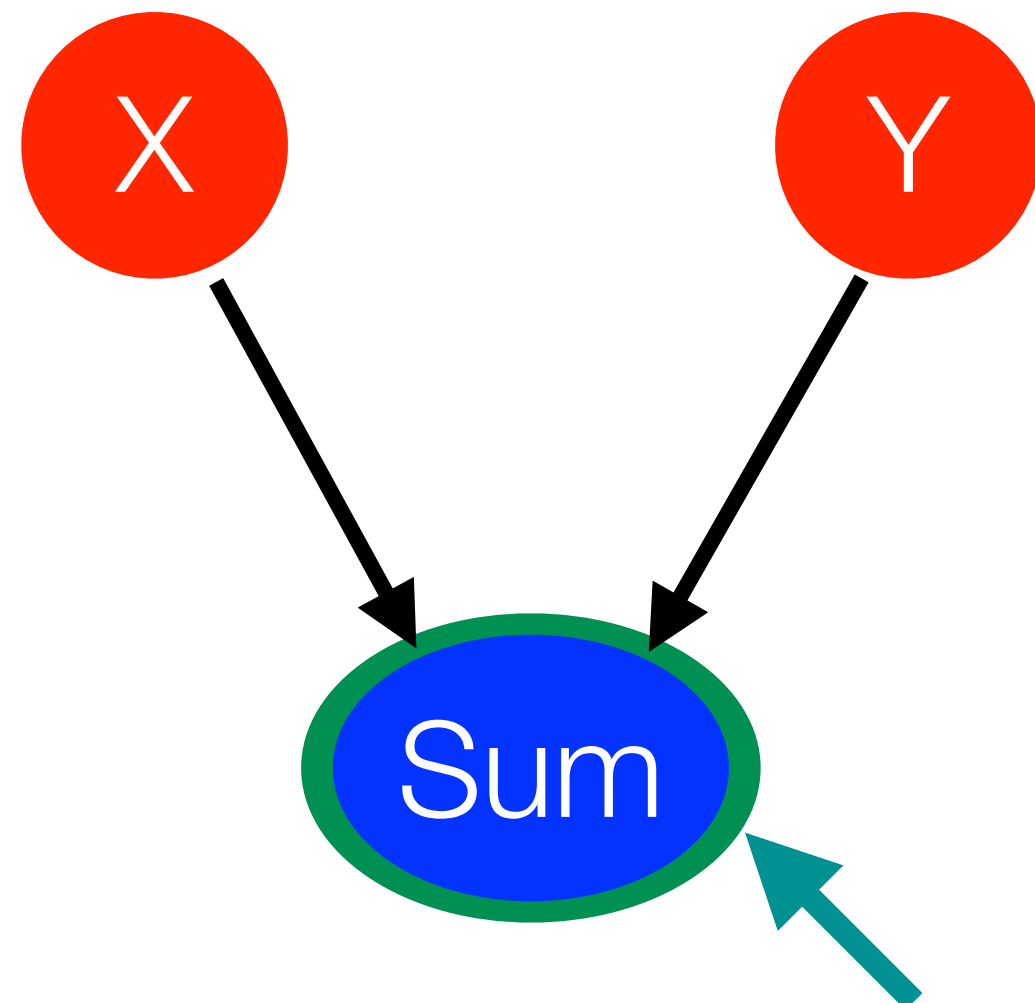
for(int i=9; i>=0; i--) {
    double sum = x_cache[i] + y_cache[i];
    grad_use(sum);
}
```

Cache Reduction Example

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

Overwritten:

Required for
Reverse:



Smallest Cache

```
double* sum_cache = new double[10];

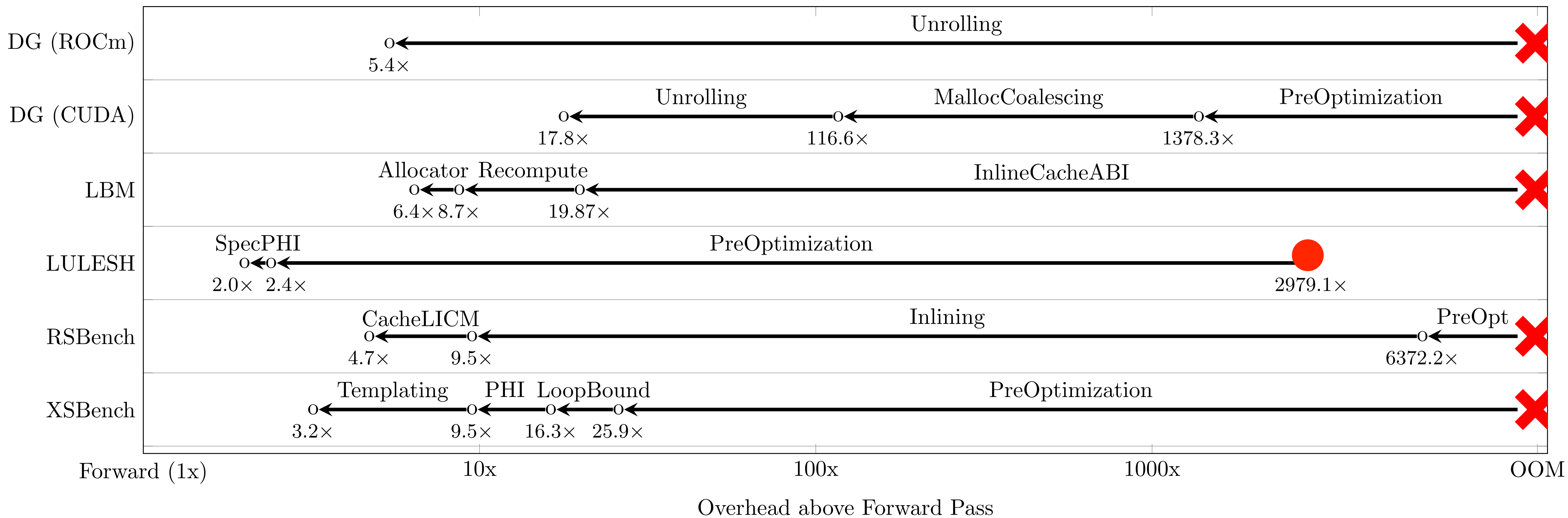
for(int i=0; i<10; i++) {
    double sum = x[i] + y[i];
    sum_cache[i] = sum;

    use(sum);
}

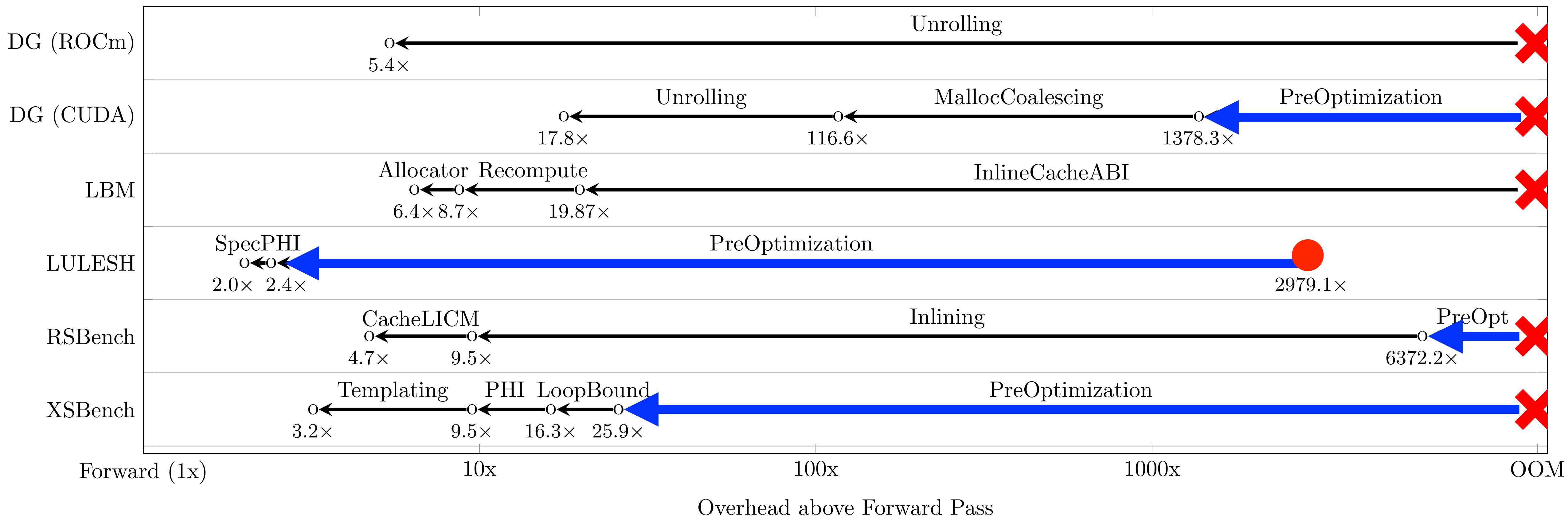
overwrite(x, y);
grad_overwrite(x, y);

for(int i=9; i>=0; i--) {
    grad_use(sum_cache[i]);
}
```

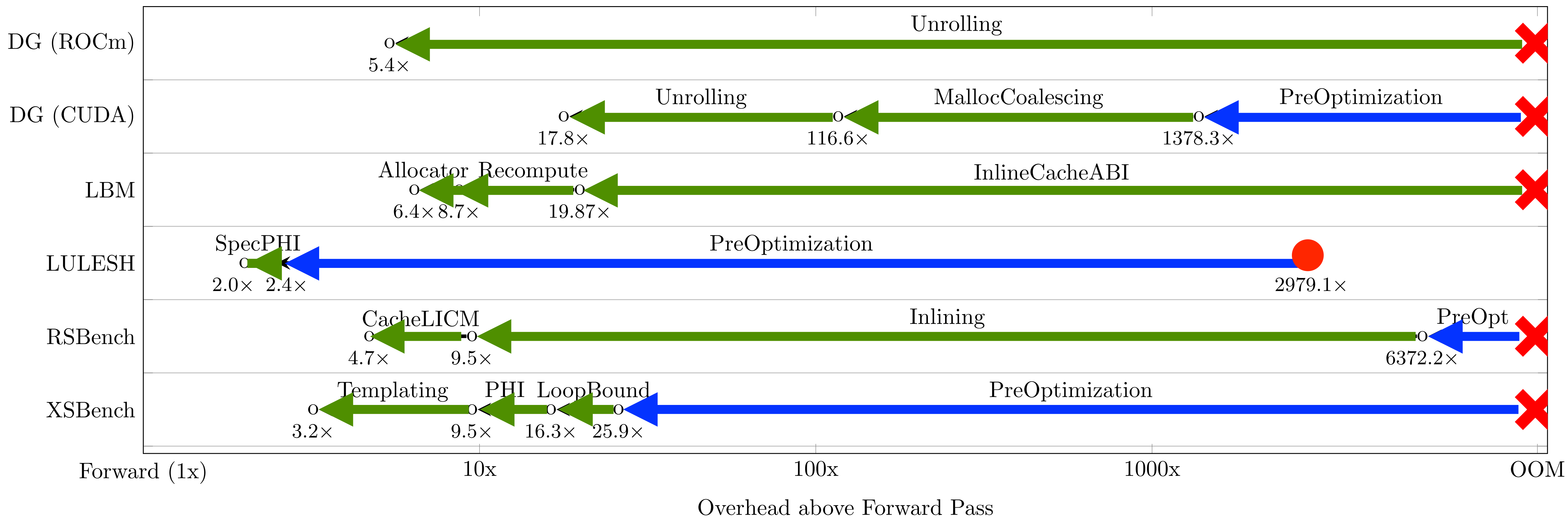
Ablation Analysis of Optimizations



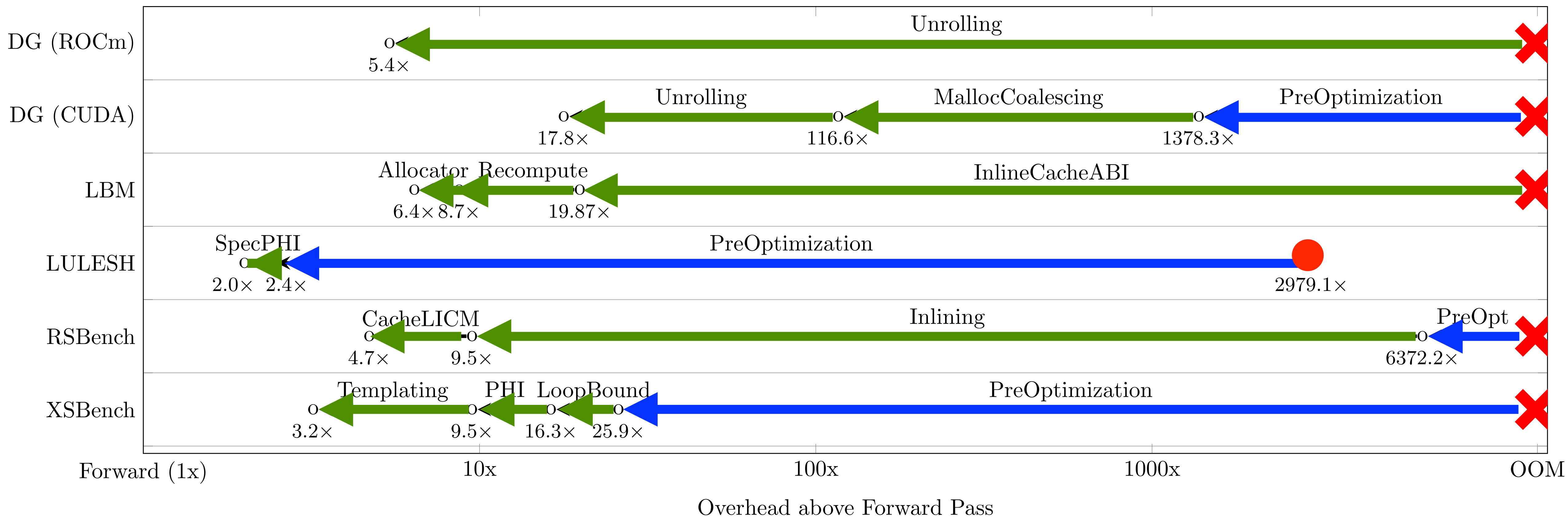
Ablation Analysis of Optimizations



Ablation Analysis of Optimizations



Ablation Analysis of Optimizations



GPU AD is Intractable Without Optimization!

What is MLIR?

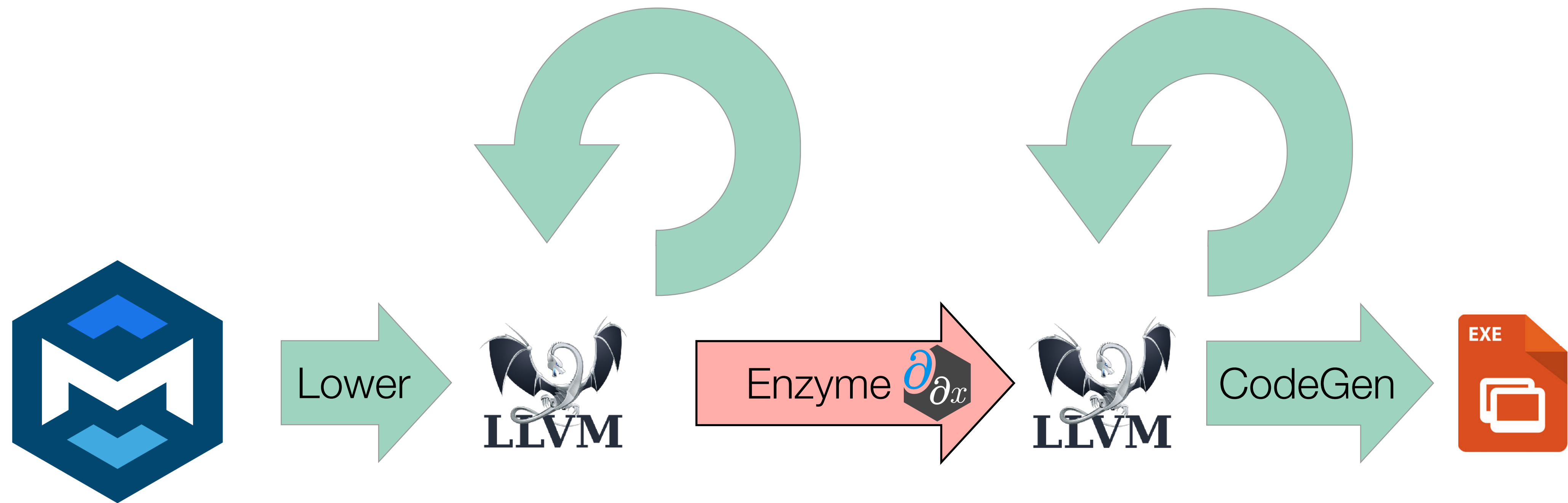


Multi-Level Intermediate Representation (MLIR)

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

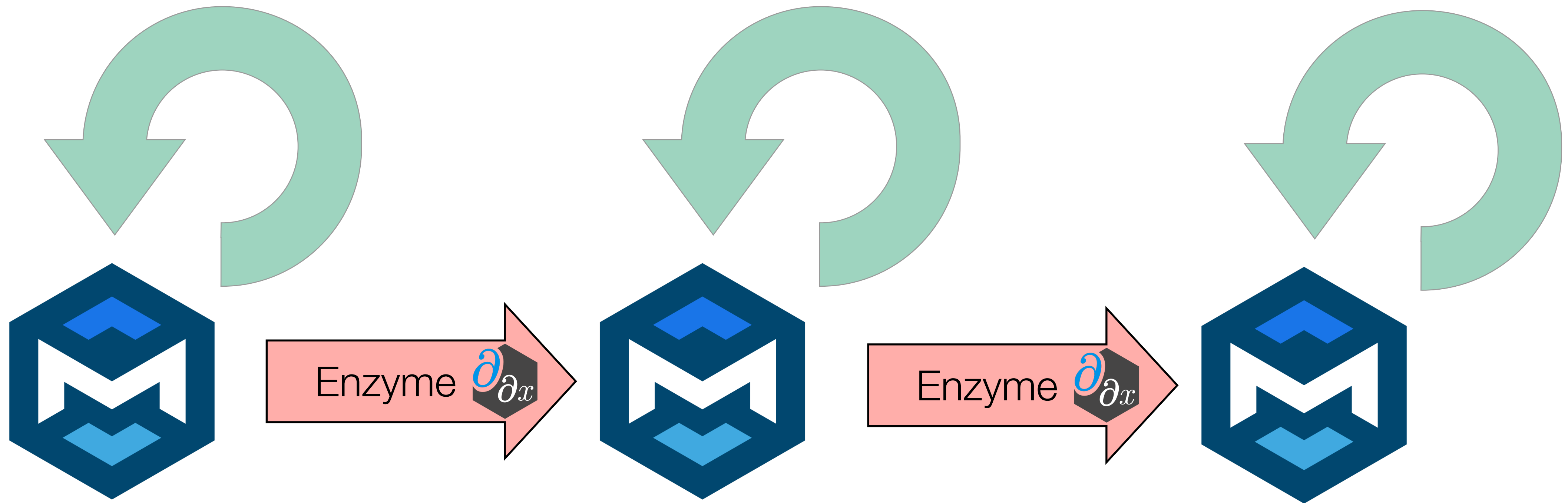
```
func @set(%arg0: memref<?xi32>, %arg1: i32) {  
  affine.for %arg2 = 0 to 10 {  
    affine.store %arg1, %arg0 [2 * %arg2] : memref<?xi32>  
  }  
  return  
}
```

Why MLIR?



Why MLIR?

“Multi-level” coordination of AD and Optimization!



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 to allow emission of different assembly and synchronization is treated as a complete optimization barrier.

```
__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

Device Code

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

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

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- 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  
}
```

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- 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  
}
```

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization
- Optimizations on primal => ***outsized impact for derivatives***

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



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

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

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

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

Often require program context

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

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

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

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

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

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

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

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

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

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

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

Often require program context

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

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

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

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

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

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

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

```
return c
```

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

$x + 0 \rightarrow x$

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

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

Often require program context

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

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

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

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

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

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

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

```
return c
```

Performance Engineering a Toy LLM

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

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

Unopt: 0.479 samples/sec

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

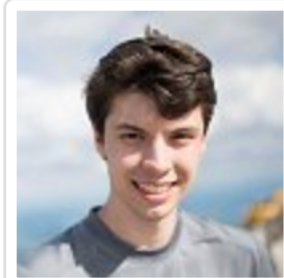
Opt: 0.736 samples/sec

Performance Engineering a Toy LLM

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



Martin Lücke
University of Edinburgh
United Kingdom



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



Michel Steuwer
Technische Universität
Berlin
Germany



Alex Zinenko
Google DeepMind



Albert Cohen
Google DeepMind
France

Mon 3 Mar

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

17:40

20m



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

Talk

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

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

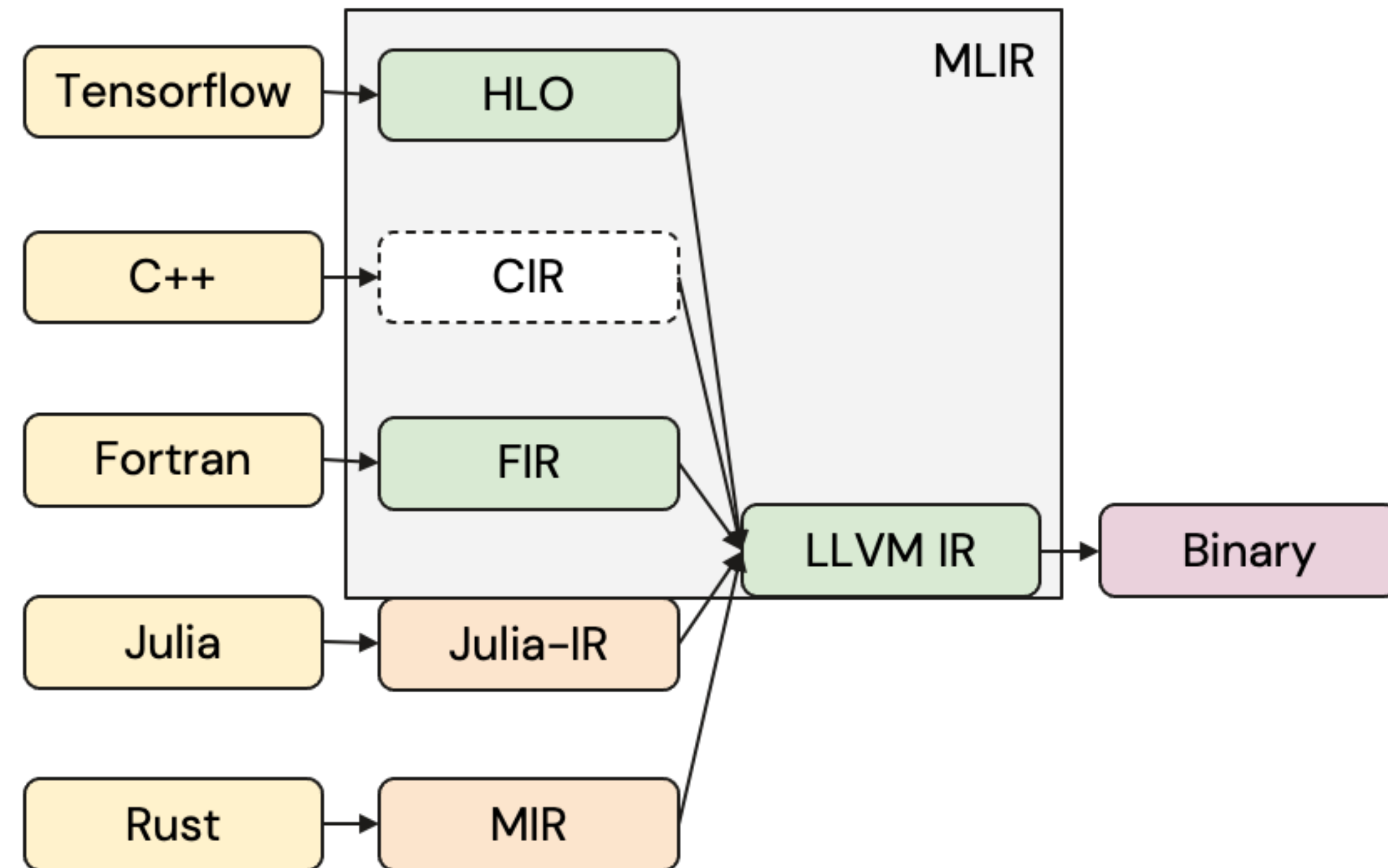
Unopt: 0.479 samples/sec

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

Opt: 0.736 samples/sec

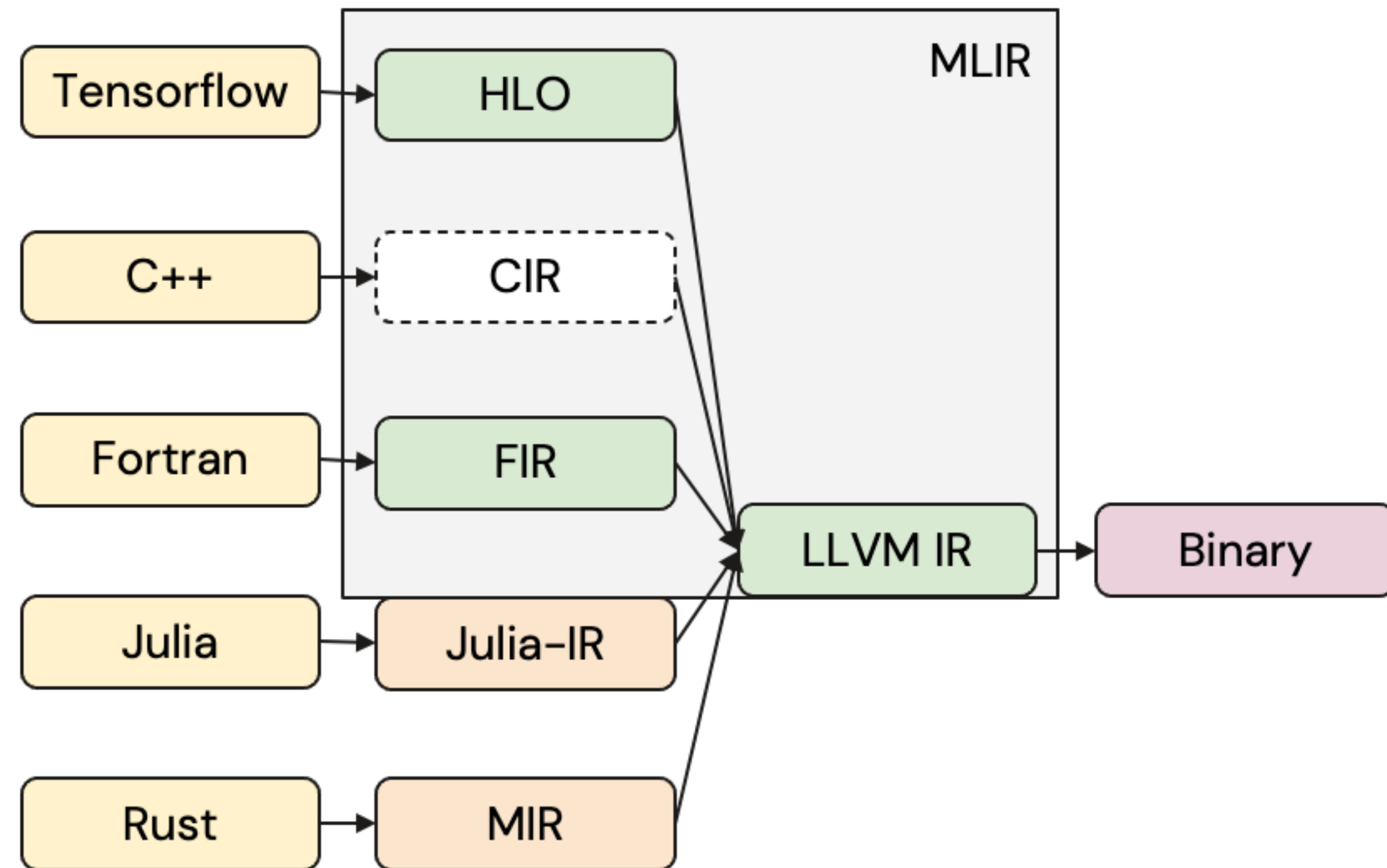
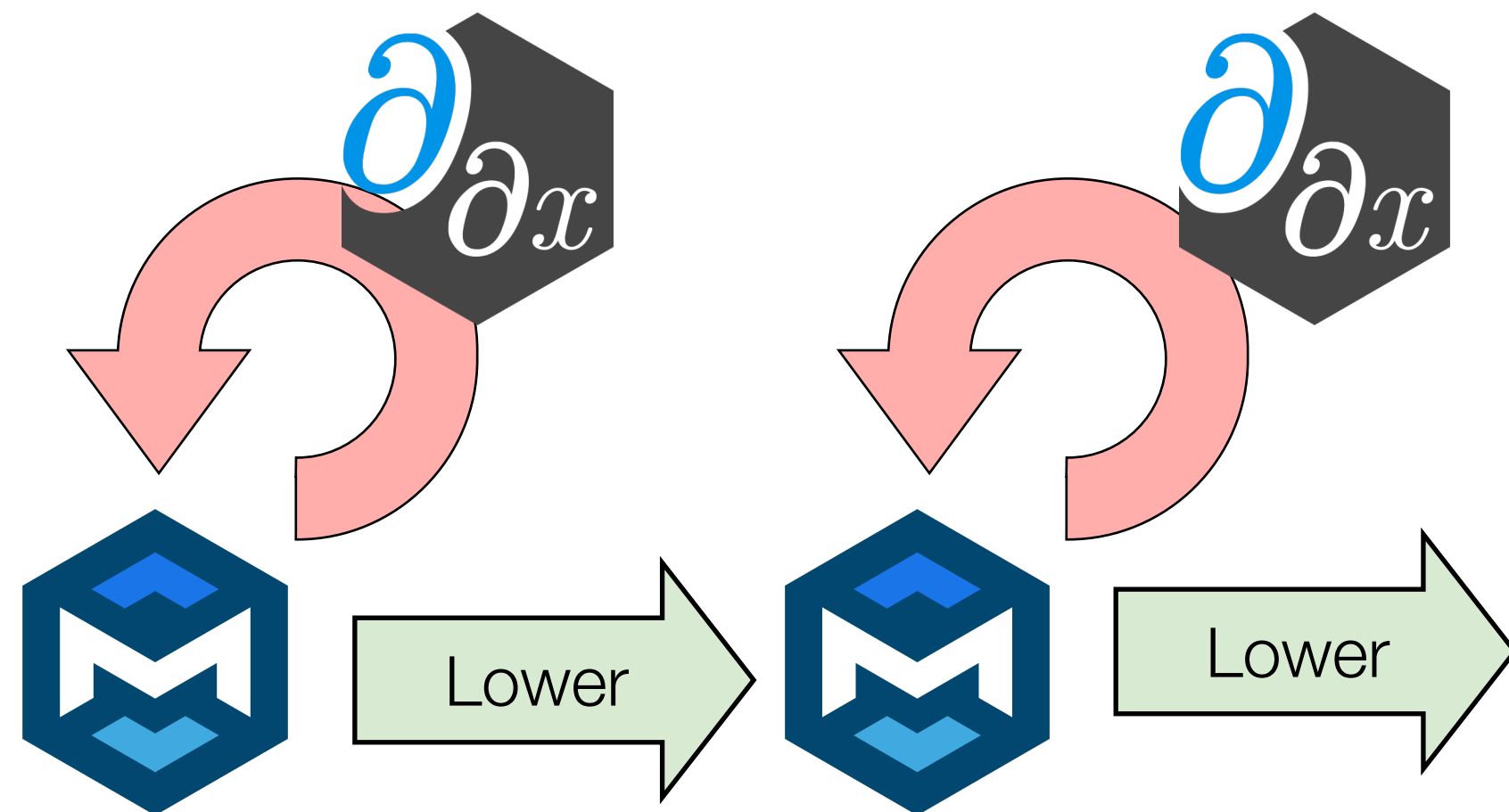
Multi-Level Differentiation

- Zoo of different MLIR dialects for various domains and optimizations
- Do we really have to write differentiation for each of the sub-abstractions again?



Multi-Level Differentiation

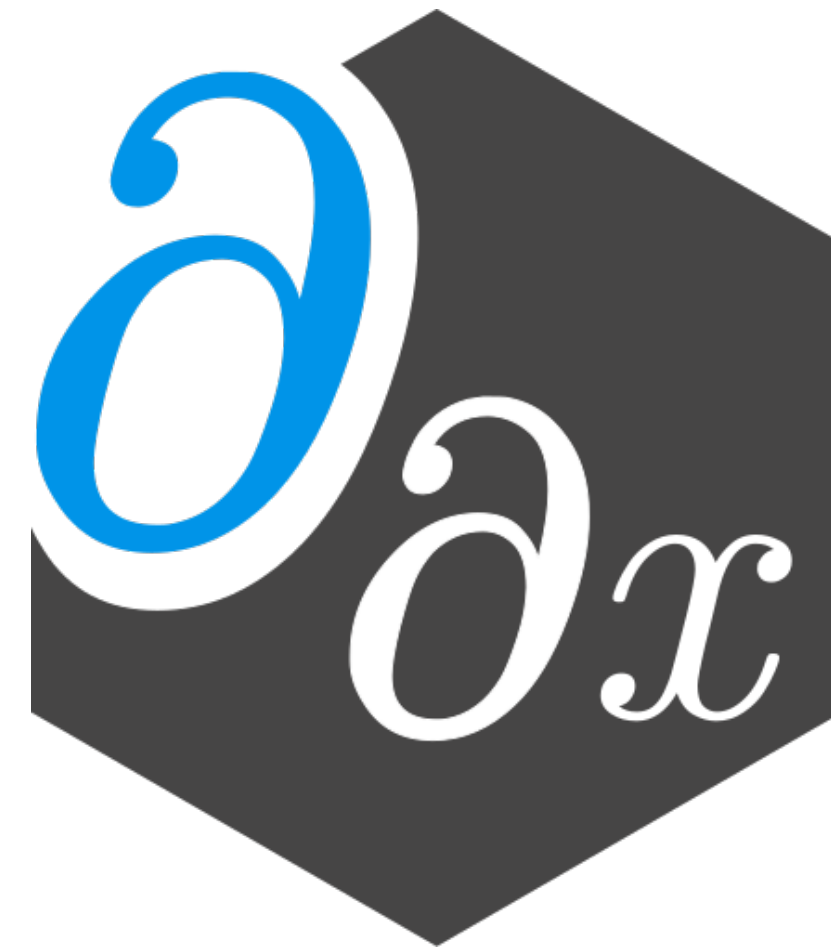
- Zoo of different MLIR dialects for various domains and optimizations
- Do we really have to write differentiation for each of the sub-abstractions again?
- No! By leveraging deferred/“multi-level” differentiation



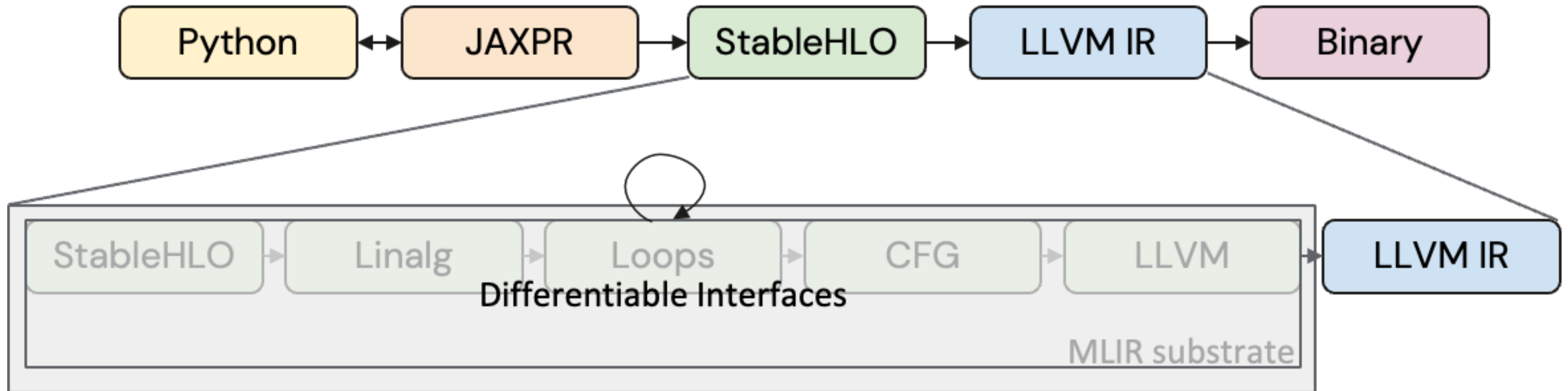
Ongoing Work



+



What is the Right Level of Abstraction for AD?

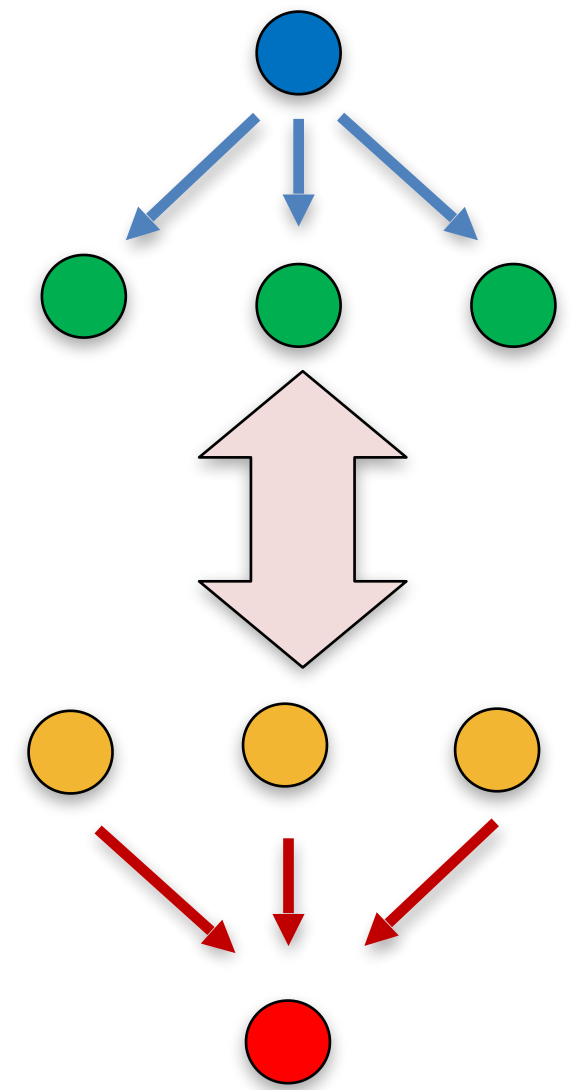


* MLIR abstractions may co-exist with each other

MLIR Enables Joint Optimization of Differentiation + Parallelism

Differentiation changes how we want to parallelize code

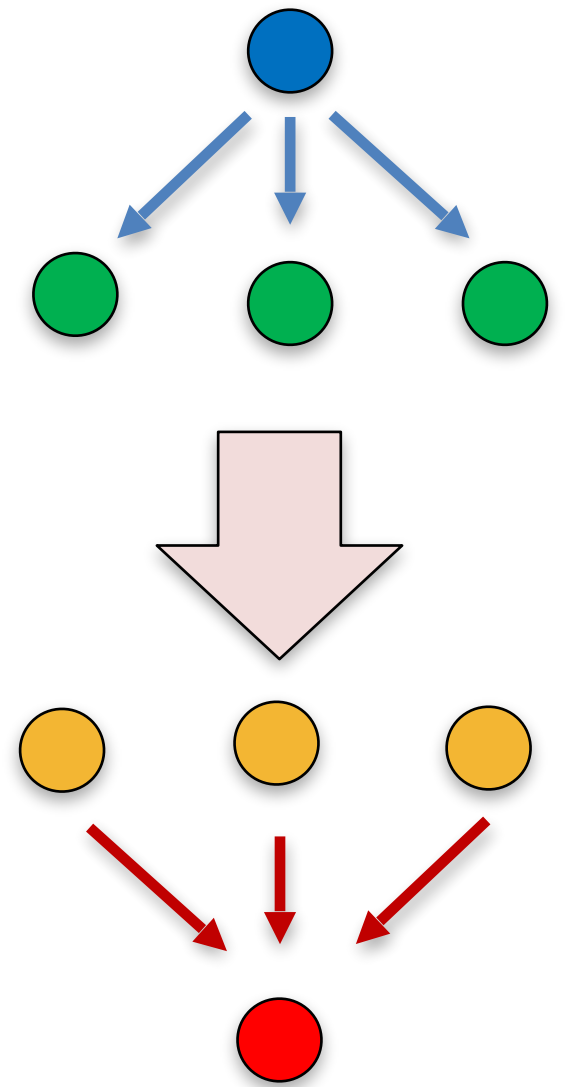
- Scatters $\leftrightarrow$ Gathers



MLIR Enables Joint Optimization of Differentiation + Parallelism

Differentiation changes how we want to parallelize code

- Scatters $\leftrightarrow$ Gathers
- Can create **race conditions**



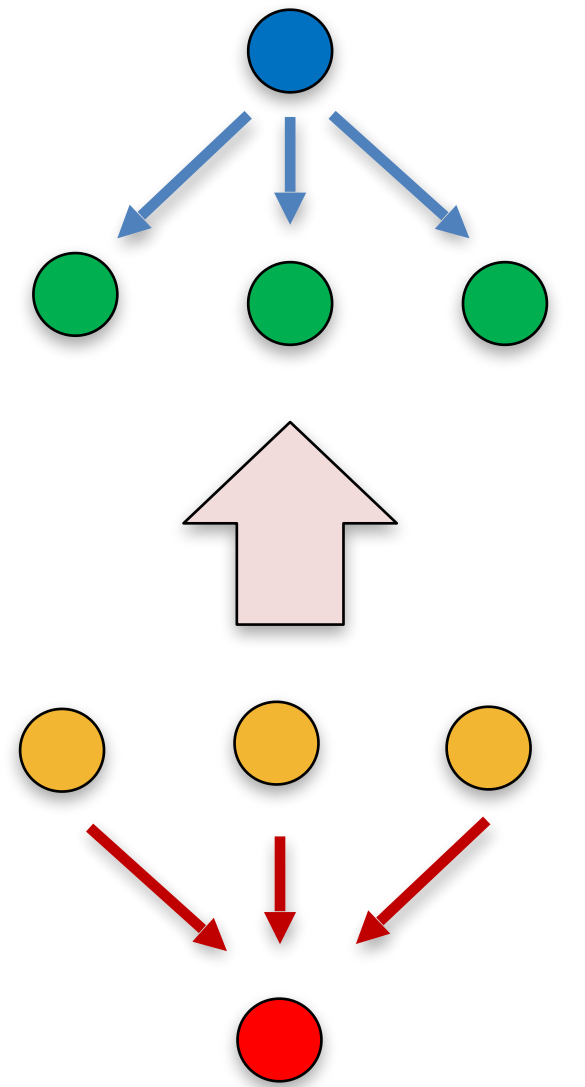
```
void set(double* ar, double val) {  
    pfor(int i=0; i<n; i++) {  
        ar[i] = val;  
    }  
    ...  
}
```

```
void grad_set(double* ar, double* d_ar) {  
    double d_val = 0;  
    pfor (int i=0; i<n; i++) {  
        d_val += d_ar[i];  
    }  
    ...  
}
```

MLIR Enables Joint Optimization of Differentiation + Parallelism

Differentiation changes how we want to parallelize code

- Scatters $\leftrightarrow$ Gathers
- Can create **race conditions**
- Serial Primal $\Rightarrow$ Parallel Derivative



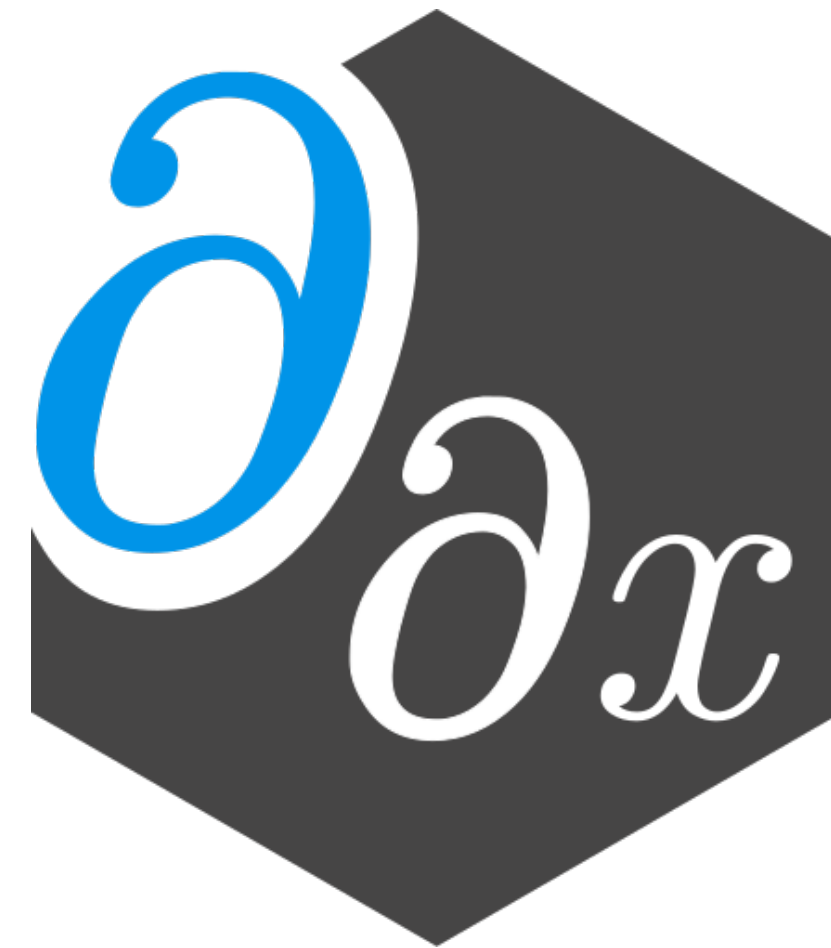
```
double sum(double* x) {  
    double S = 0.0;  
    for (int i = 0; i < N; i++) {  
        S += x[i] * x[i];  
    }  
    return S;  
}
```

```
void grad_sum(double* x, double* d_x,  
              double d_S) {  
    pfor (int i = 0; i < N; i++) {  
        d_x[i] += 2.0 * x[i] * d_S;  
    }  
}
```

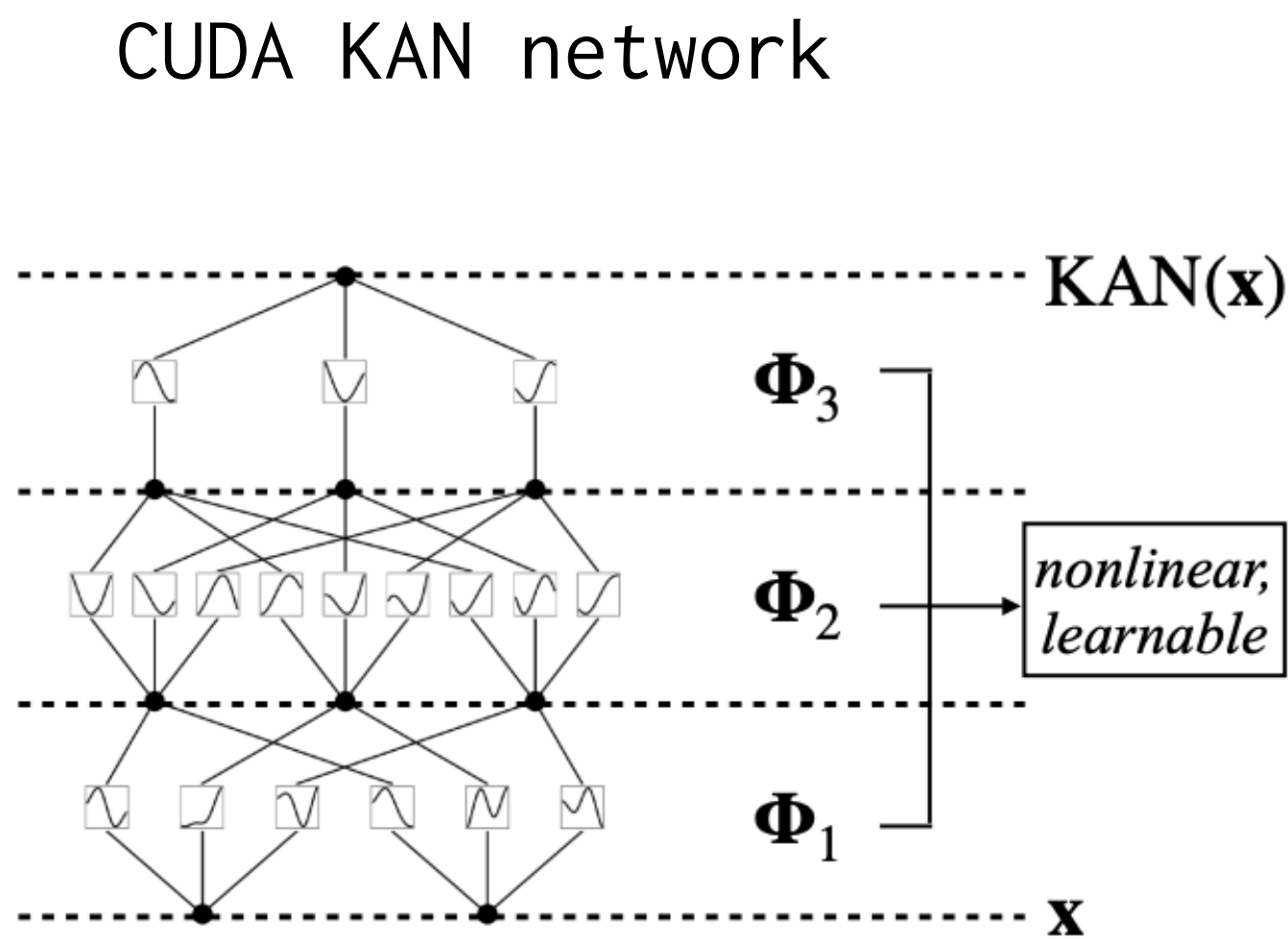
Brief Results



+



EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)



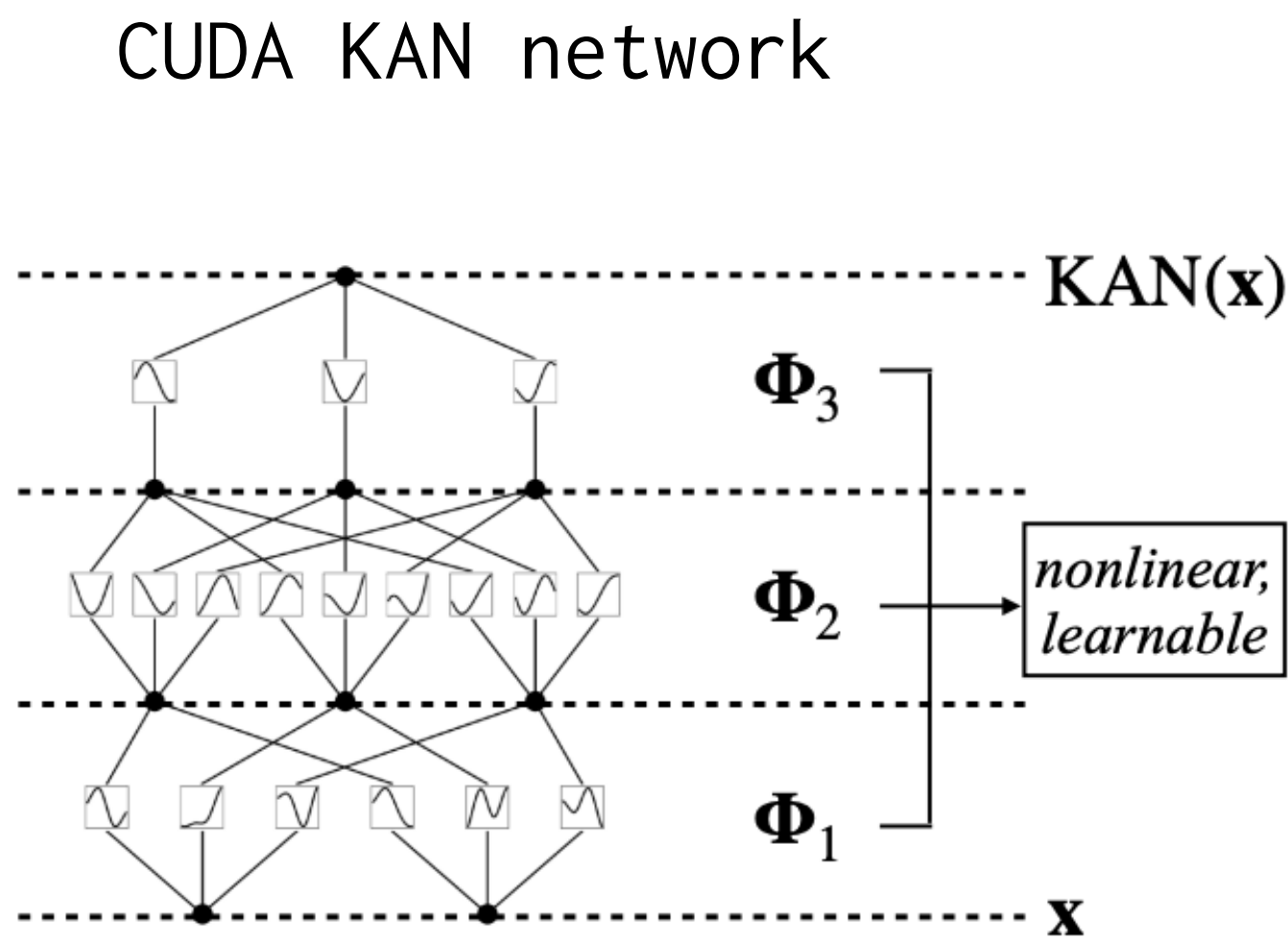
Forward (regular Julia)
47.586 us (248 allocations)
234.233 us (1022 allocations)
134.028 us (668 allocations)

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

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

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

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)



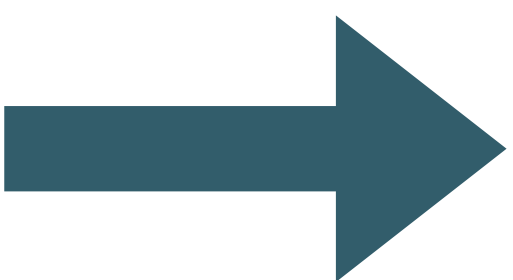
Forward (regular Julia)
47.586 us (248 allocations)
234.233 us (1022 allocations)
134.028 us (668 allocations)

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

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

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

2.14x speedup
(Primal)



13.57x speedup
(Derivative)



Takeaways

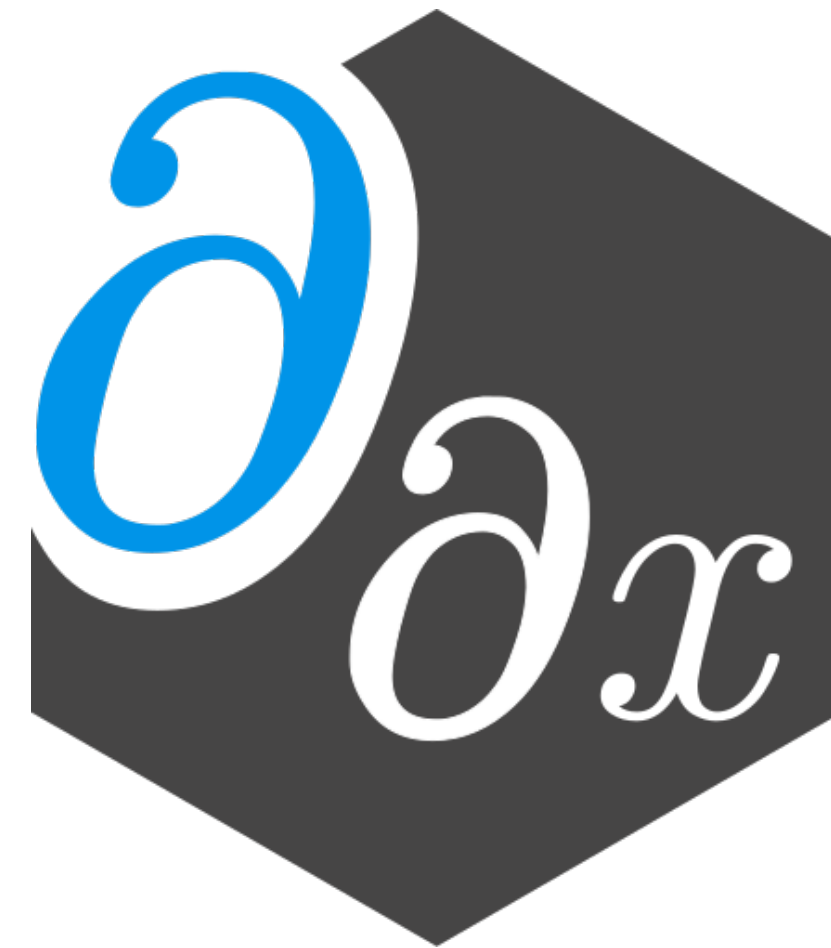
- Compilers Make Differentiation Fast and Easy to use
 - Key to this is interaction with Optimization
- MLIR enables preserving and optimizing high-level structure
- EnzymeMLIR enables differentiation to combine with high-level optimization
 - Extensible for use with any MLIR dialect
 - Enables significant performance gains when combined with linear algebra, parallel optimizations
- All open source ([GitHub.com/EnzymeAD/Enzyme](https://github.com/EnzymeAD/Enzyme) ; [GitHub.com/EnzymeAD/Enzyme-JaX](https://github.com/EnzymeAD/Enzyme-JaX) ; [GitHub.com/EnzymeAD/Reactant.jl](https://github.com/EnzymeAD/Reactant.jl))



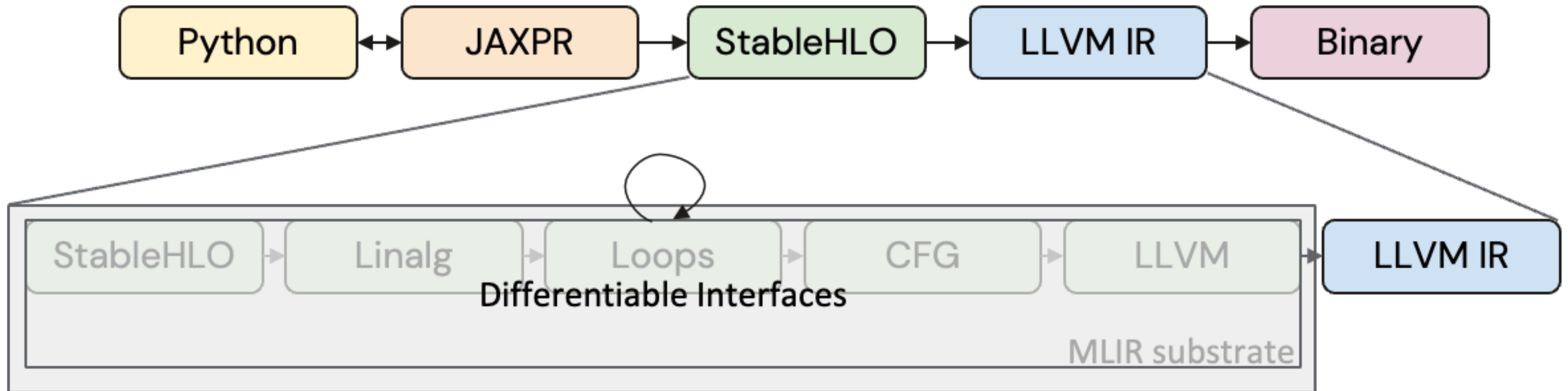
Integrating a Dialect with Enzyme



+



Differentiation Rules as Interfaces on Operations



* MLIR abstractions may co-exist with each other

Operation Interfaces

Interface Methods To Implement

- `createForwardModeTangent`
generates the IR for forward tangent(s)
- `createReverseModeAdjoint`
generates the IR for backward tangent(s)
- `cacheValues`
generates the IR to store values from the primal computation needed for the tangent
- `createShadowValues`
generates the IR to allocate and initialize shadow memory needed for the tangent

Example: Float Scalar/Vector Multiplication

```
d(res) = addf(mulf(LHS, d(RHS)),  
              mulf(d(LHS), RHS));
```

```
d(LHS) += mulf(pop(cache[0]), d(res))  
d(RHS) += mulf(pop(cache[1]), d(res))
```

```
push(cache[0], RHS)  
push(cache[1], LHS)
```

Noop

Control Flow Interfaces

Interface Methods To Implement

- `createWithShadows`
creates a copy of the operation with placeholders for shadow values in data flow
- `createReverseControlFlow`
creates a copy of the operation with reverted control flow and placeholders for body

Example: “for” loop

- => `scf.for i=0..N args(a, b)`
=> `scf.for i=0..N args(a, da, b, db)`
- => `scf.for i=0..N`
=> `scf.for ii=0..N {`
 `i=N-ii-1`
 `}`

Type Interfaces

Interface Methods To Implement [only needed for reverse mode]

- `getShadowType`
returns mutable type suitable for storing in shadow memory. If mutable, can return self.
- `createNullValue`
generates the IR initializing the a null shadow of this type
- `createAddToOp`
generates the IR adding a value of this type to the shadow

Example: Float

```
memref<self>
```

```
%shadow = memref.alloca()  
%shadow[] = constant(cast<..>(0.0))
```

```
%shadow[] += val
```

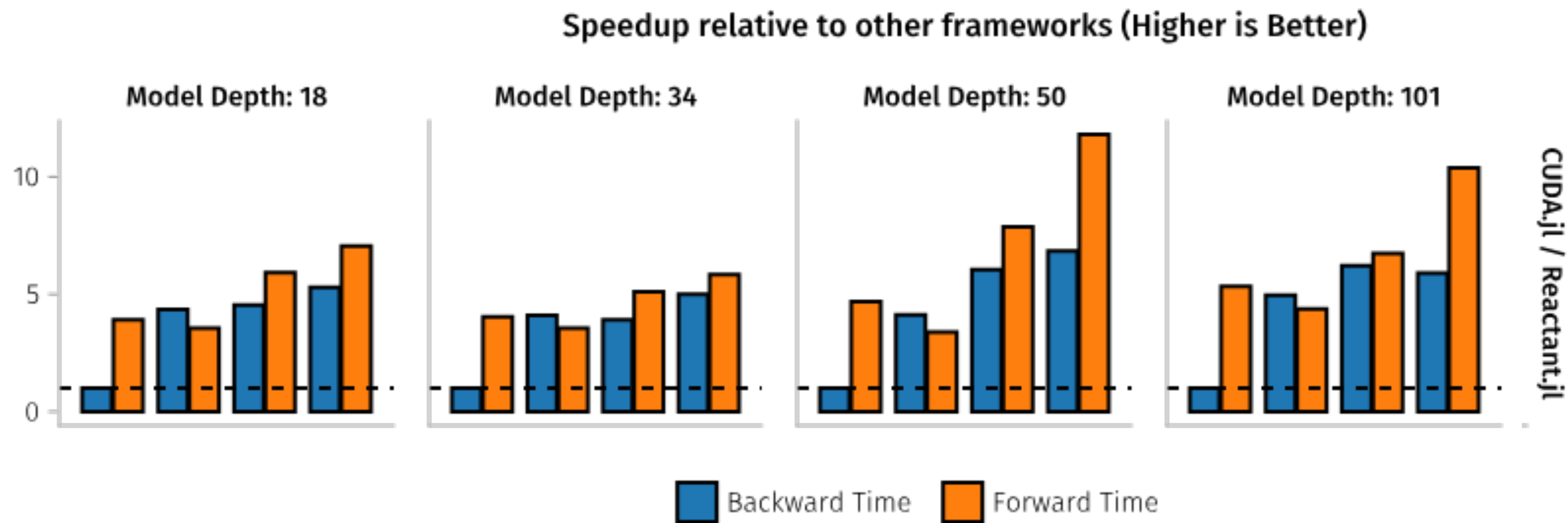
Concise AD Definitions in ODS/Tablegen

```
def : MLIRDerivative<"arith", "DivFOp", (Op $x, $y),  
    // Reverse mode.  
    [  
        (DivF (DiffeRet), $y),  
        (NegF (MulF (DivF (DiffeRet), $y), (DivF $x, $y)))  
    ],  
    // Forward mode (defaults to sum of reverse mode in absence).
```

Concise AD Definitions in ODS/Tablegen

```
def : MLIRDerivative<"arith", "DivFOp", (Op $x, $y),
    // Reverse mode.
    [
        (DivF (DiffeRet), $y),
        (NegF (MulF (DivF (DiffeRet), $y), (DivF $x, $y)))
    ],
    // Forward mode (defaults to sum of reverse mode in absence).
    (DivF (SubF (SelectIfActive $x, (MulF (Shadow $x), $y),
        (ConstantFP<"0", "arith", "ConstantOp"> $x))),
        (SelectIfActive $y, (MulF (Shadow $y), $x),
        (ConstantFP<"0", "arith", "ConstantOp"> $y)))),
    (MulF $y, $y)),
    // Activity flow between $x,$y and the result without storage.
    [($x DiffeRet "0"), ($y DiffeRet "0")],
    >;
```

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)



Case Study: ReLU3

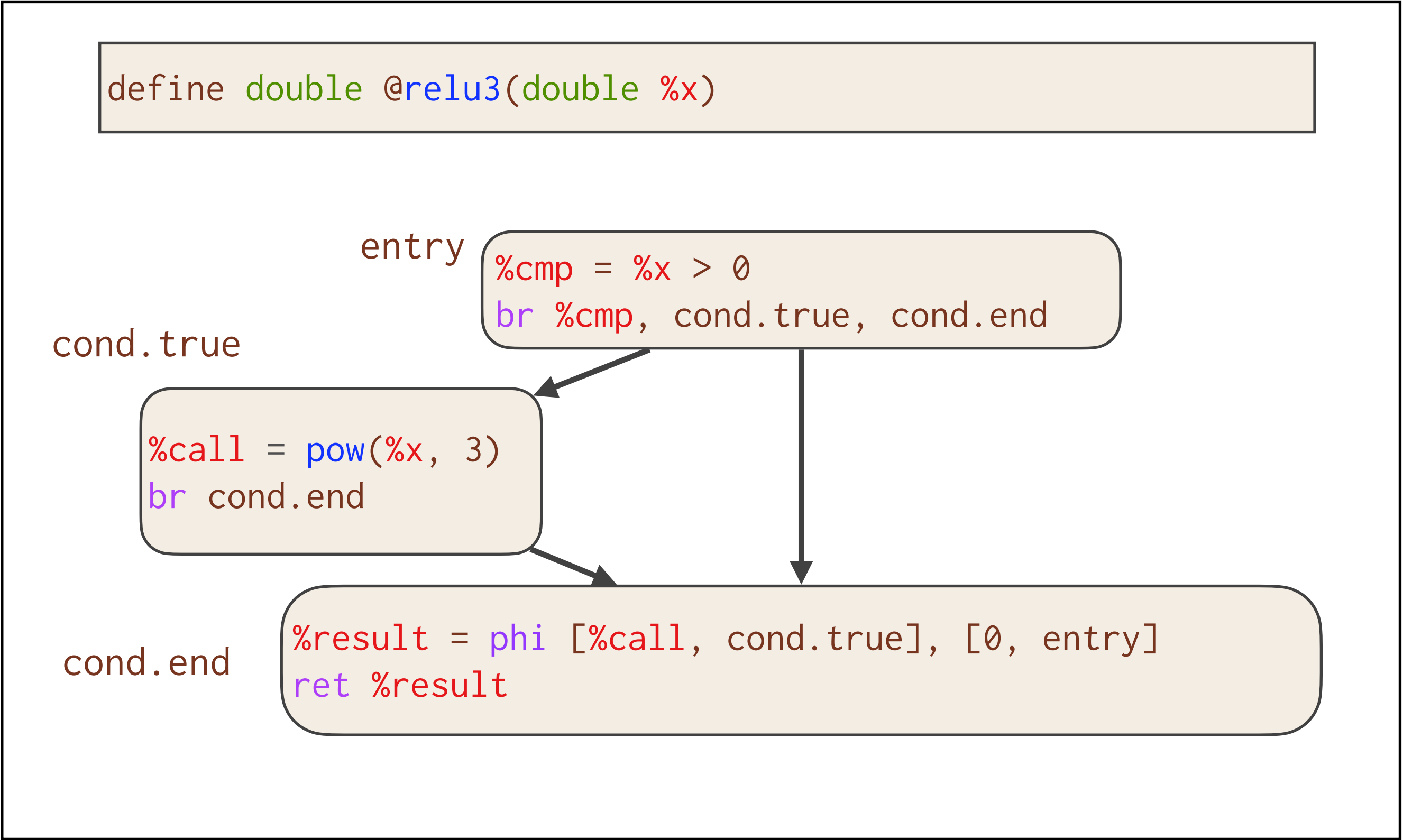
C Source

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

Enzyme Usage

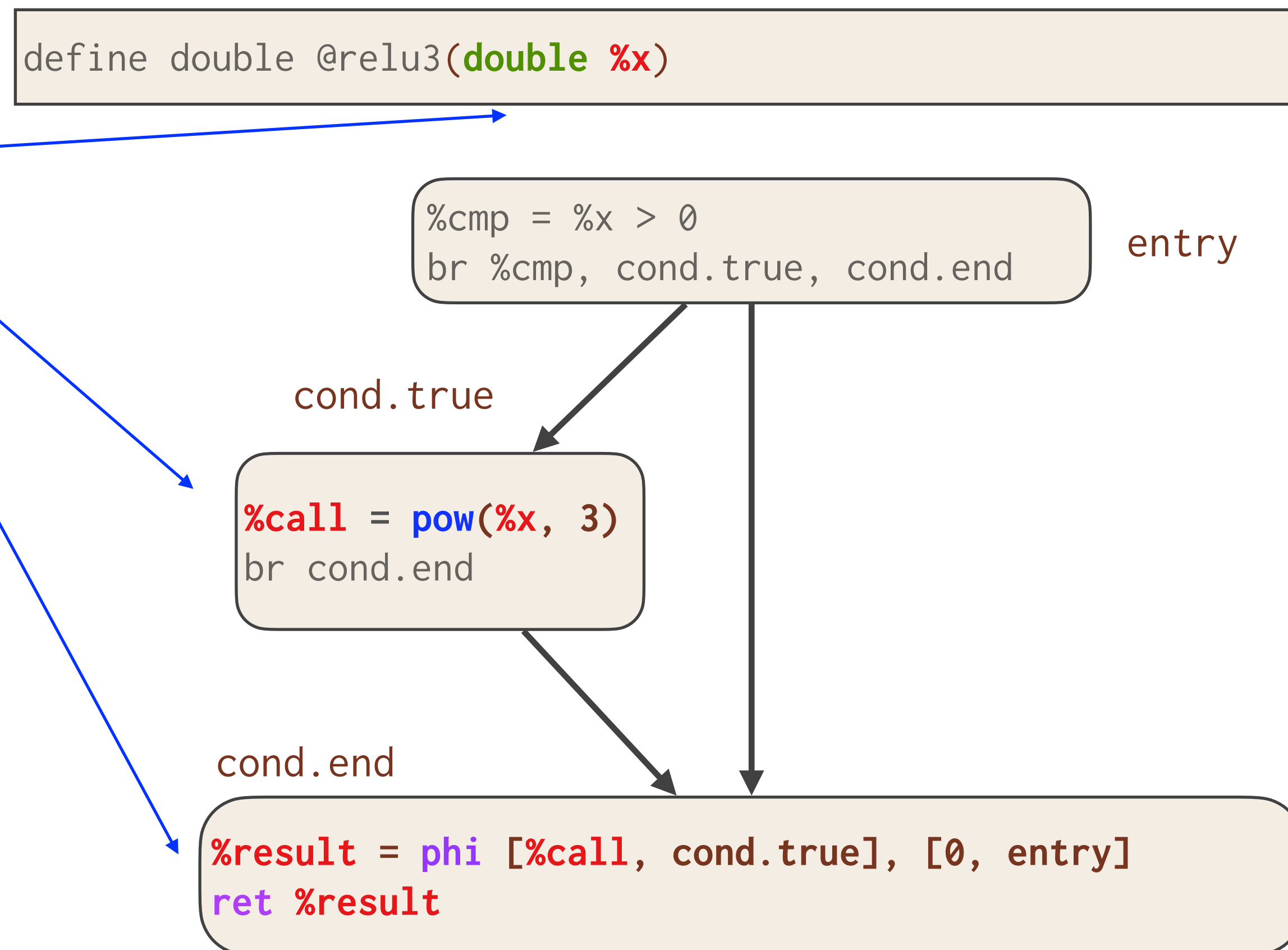
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

LLVM

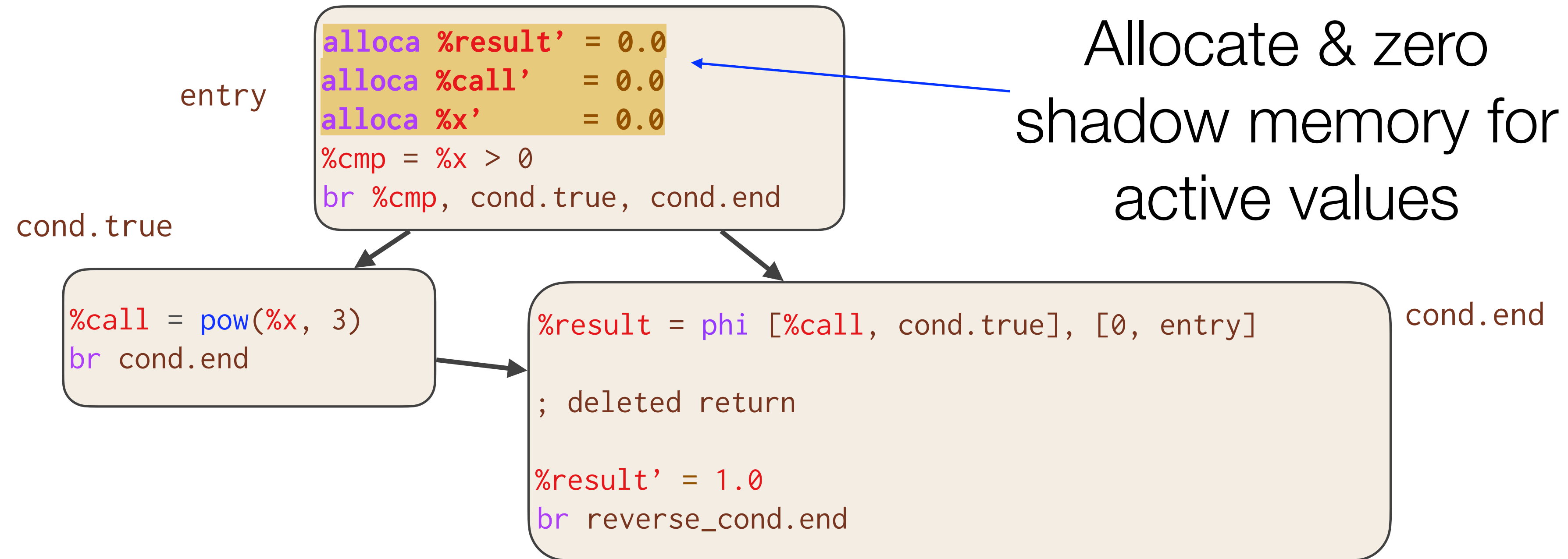


Case Study: ReLU3

Active Instructions

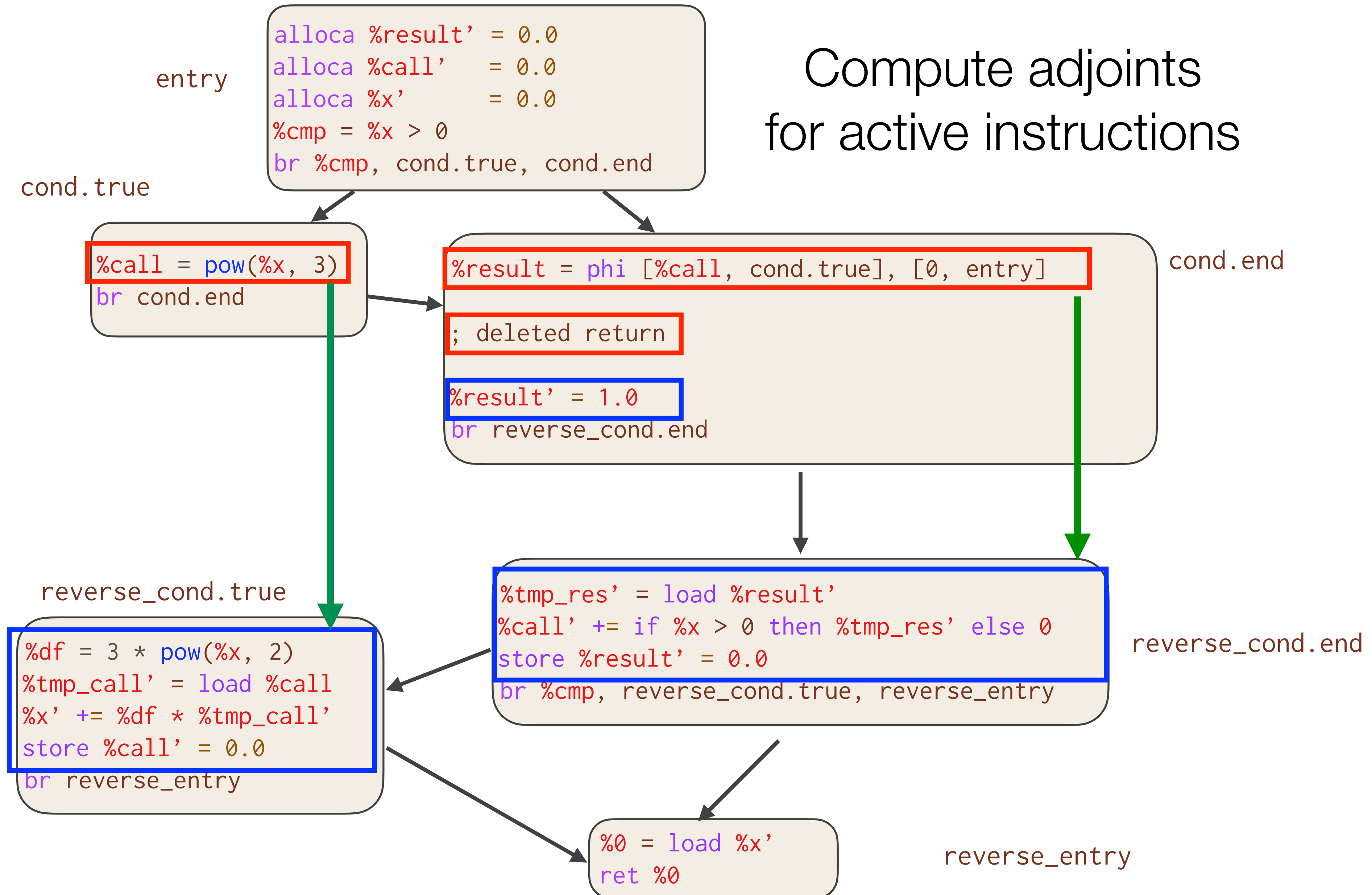


```
define double @diffe_relu3(double %x, double %differet)
```



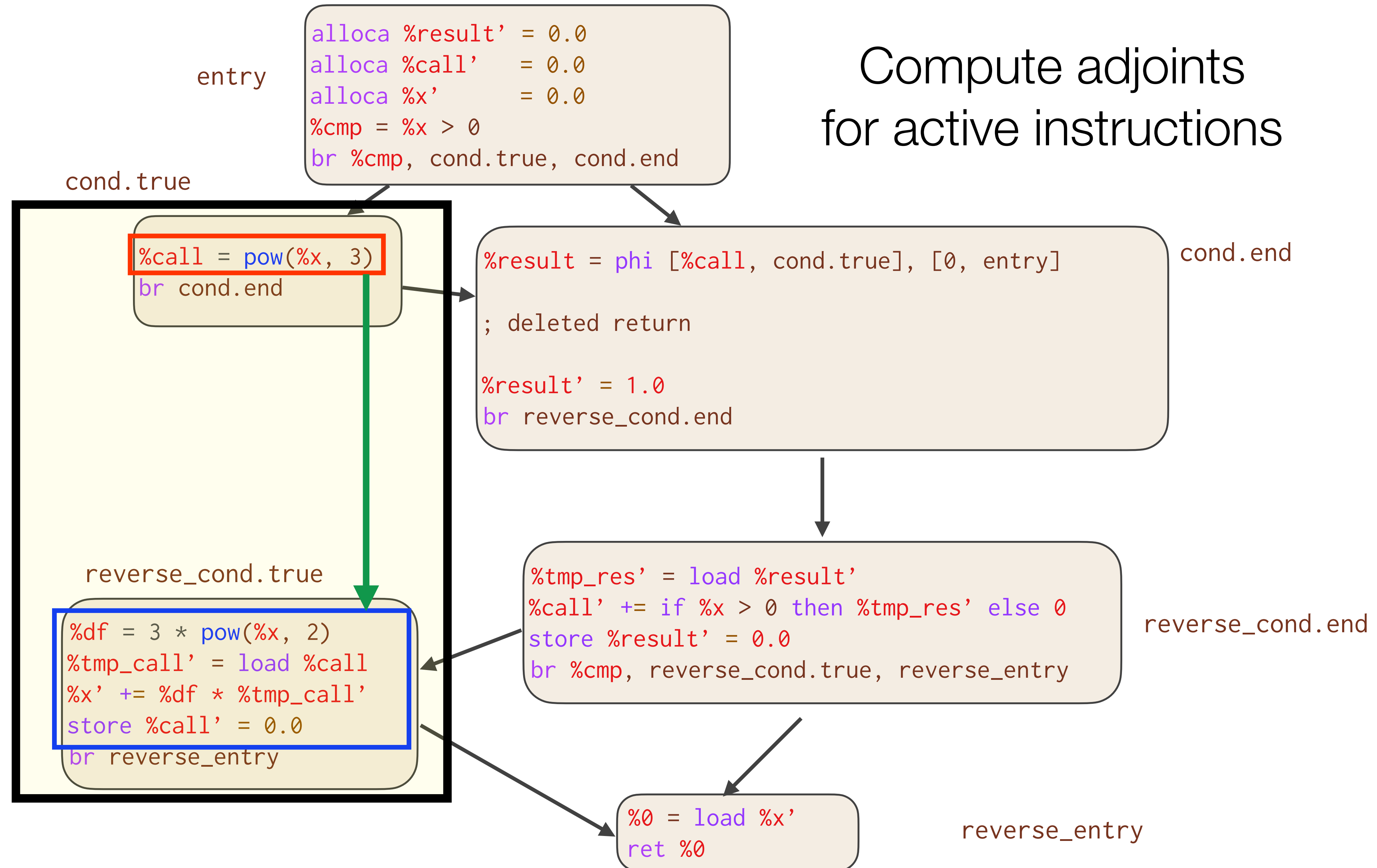
```
define double @diffe_relu3(double %x, double %differet)
```

Compute adjoints
for active instructions



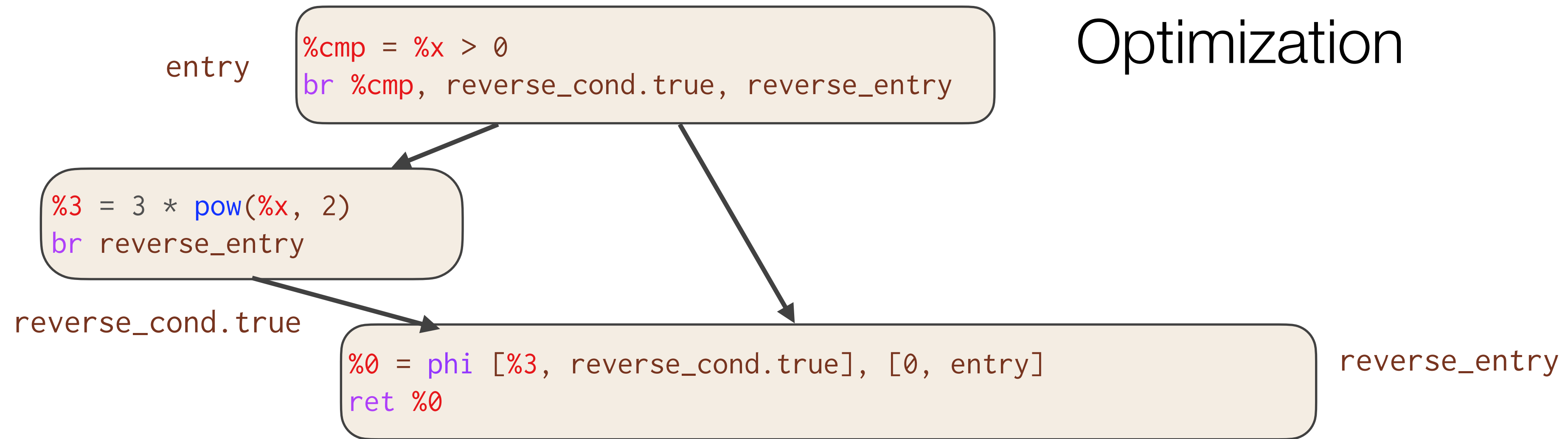
```
define double @diffe_relu3(double %x, double %differet)
```

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;  
}
```

