Enzyme-MLIR: Early Experiments on multi-level automatic differentiation


Martin Eppert


Jacob Peng


Ludger Paehler


Alex Zinenko


William S. Moses

wsmoses@illinois.edu

MLIR Summit

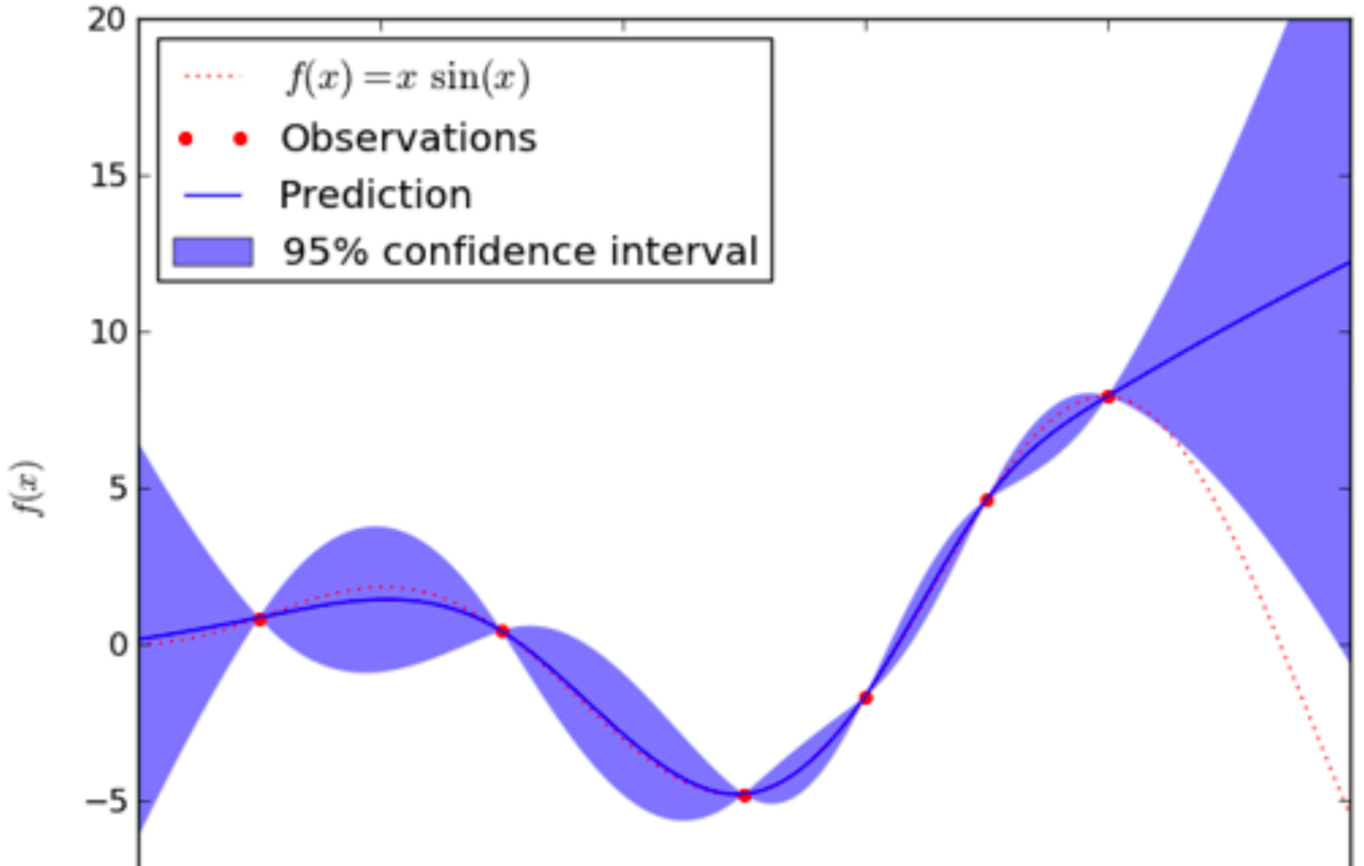
Oct 9, 2023

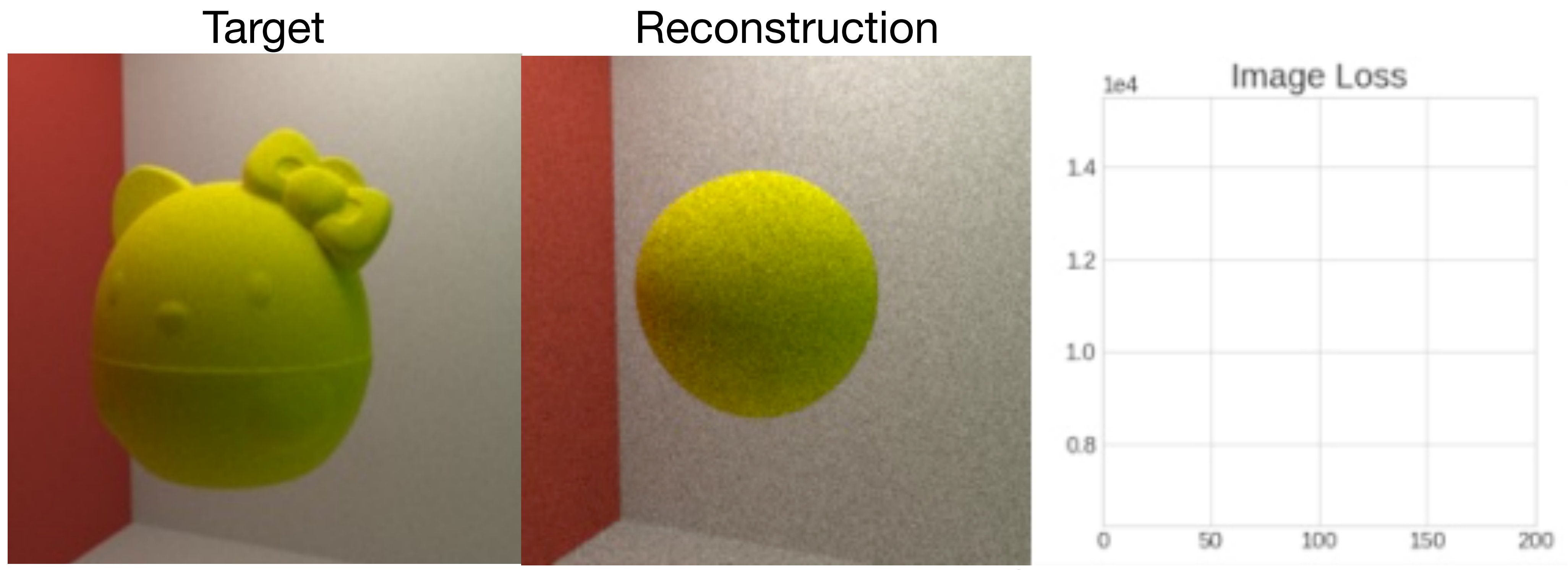
AP Calculus: Revisited

- Derivatives compute the rate of change of a function's output with respect to input(s)

$$\mathrm{f}'(\mathrm{x}) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

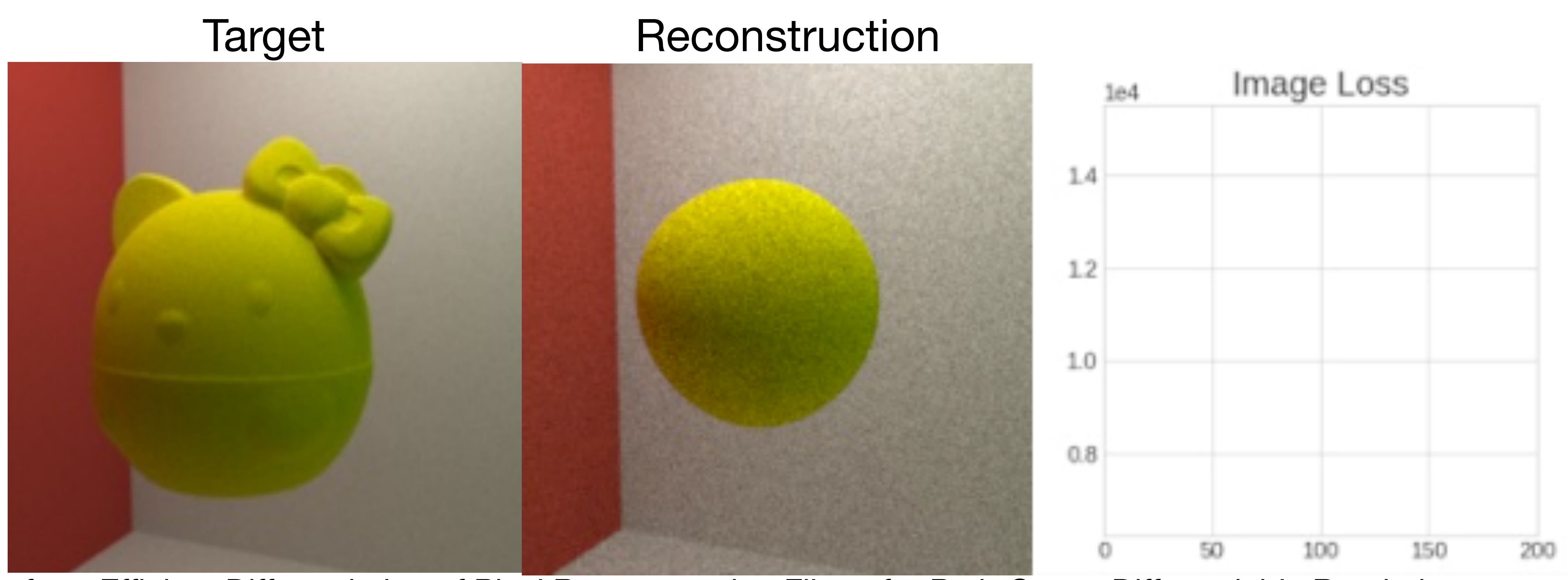
- Derivatives are used widely across science
  - Machine learning (back-propagation, Bayesian inference)
  - Scientific computing (modeling, simulation, uncertainty quantification)



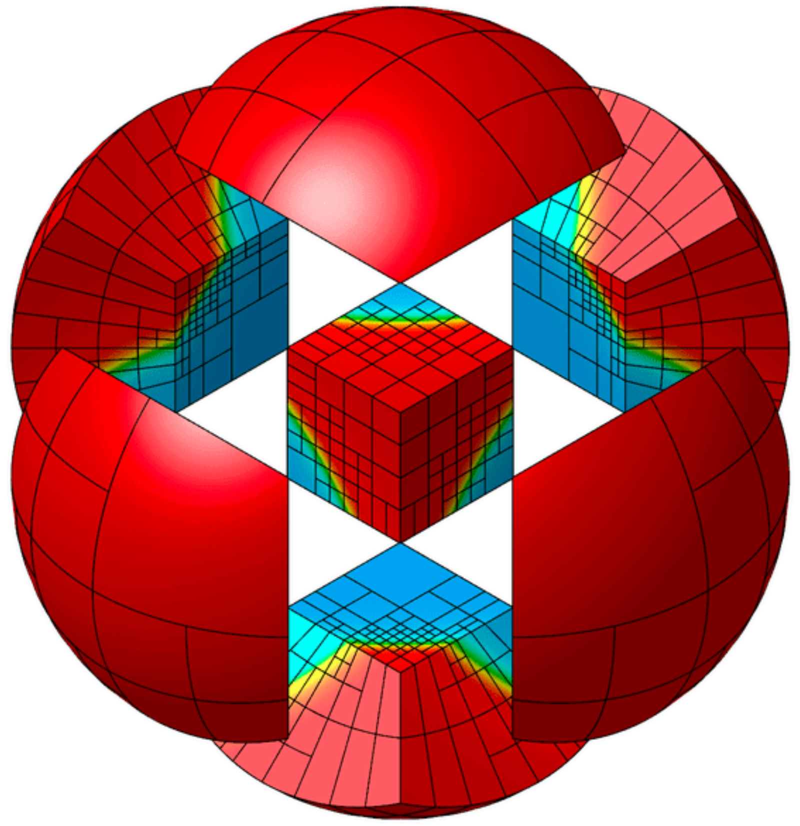


from Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering, SIGGRAPH Asia 2022, Zihan Yu et al

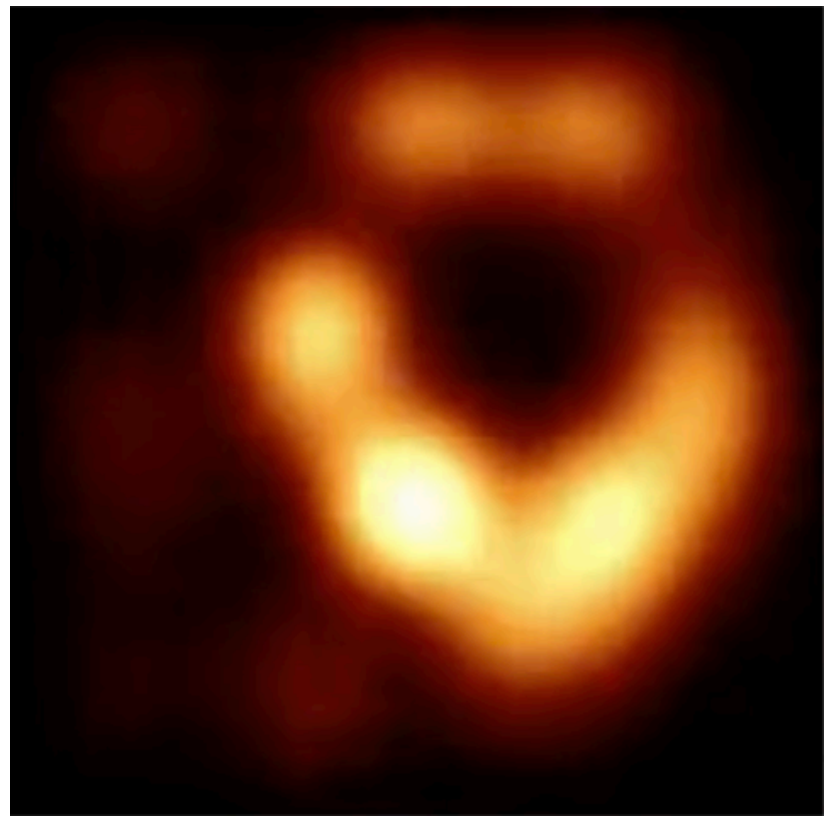
AD-Powered Applications



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

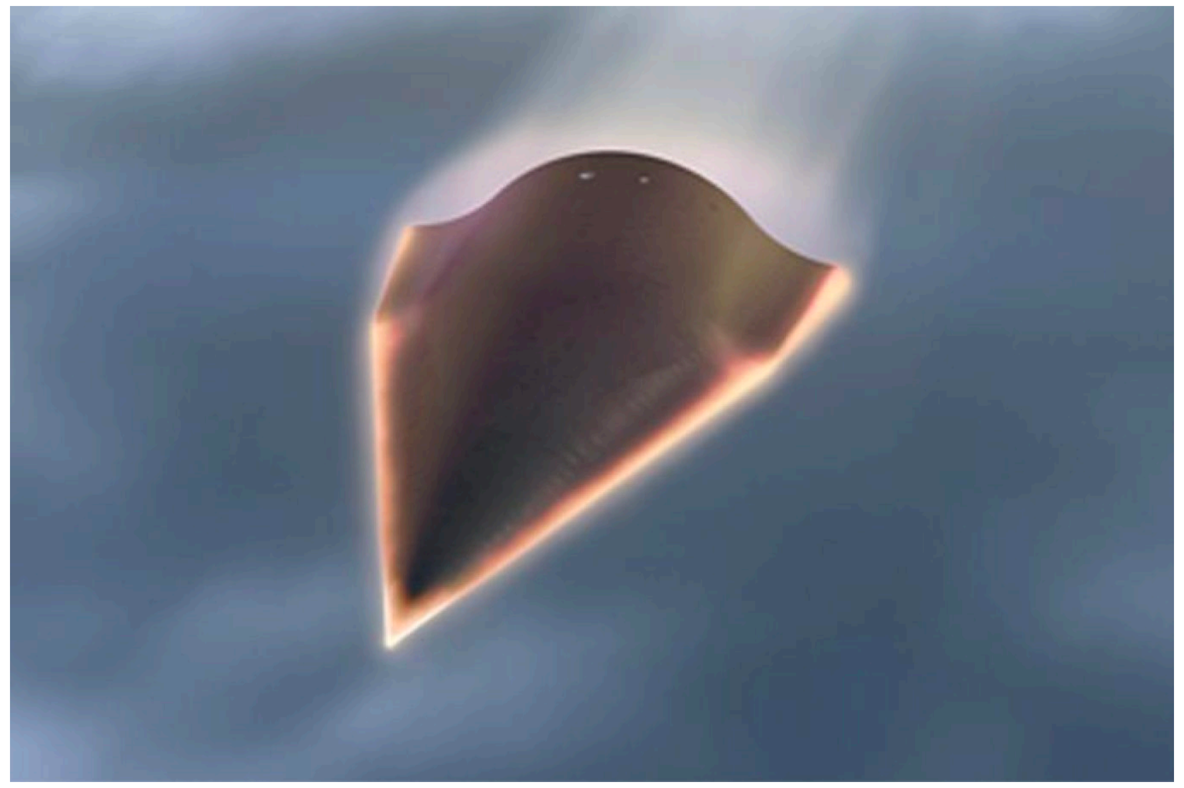


from MFEM Team at LLNL

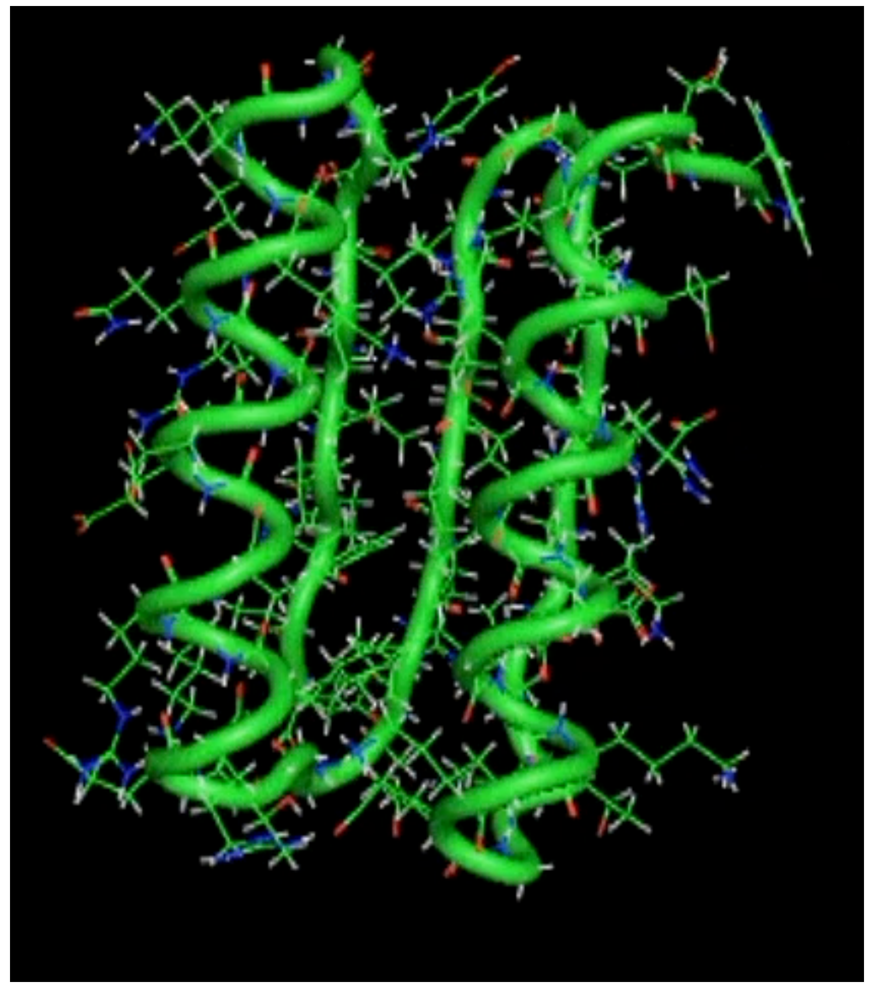


from Comrade: High Performance Black-Hole Imaging JuliaCon 2022, Paul Tiede (Harvard)

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[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)
```

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;
}
```

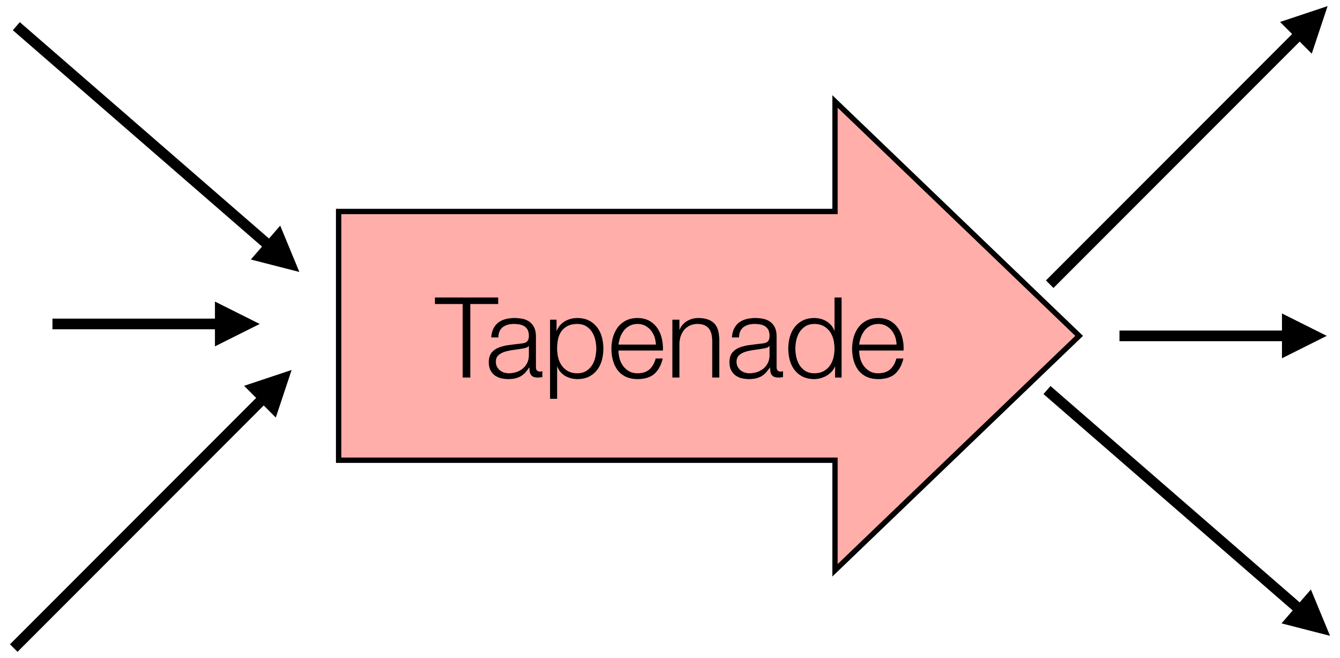
```
adept::Stack stack;
adept::adouble inp = 3.14;

// 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

```
// myfile.h
```

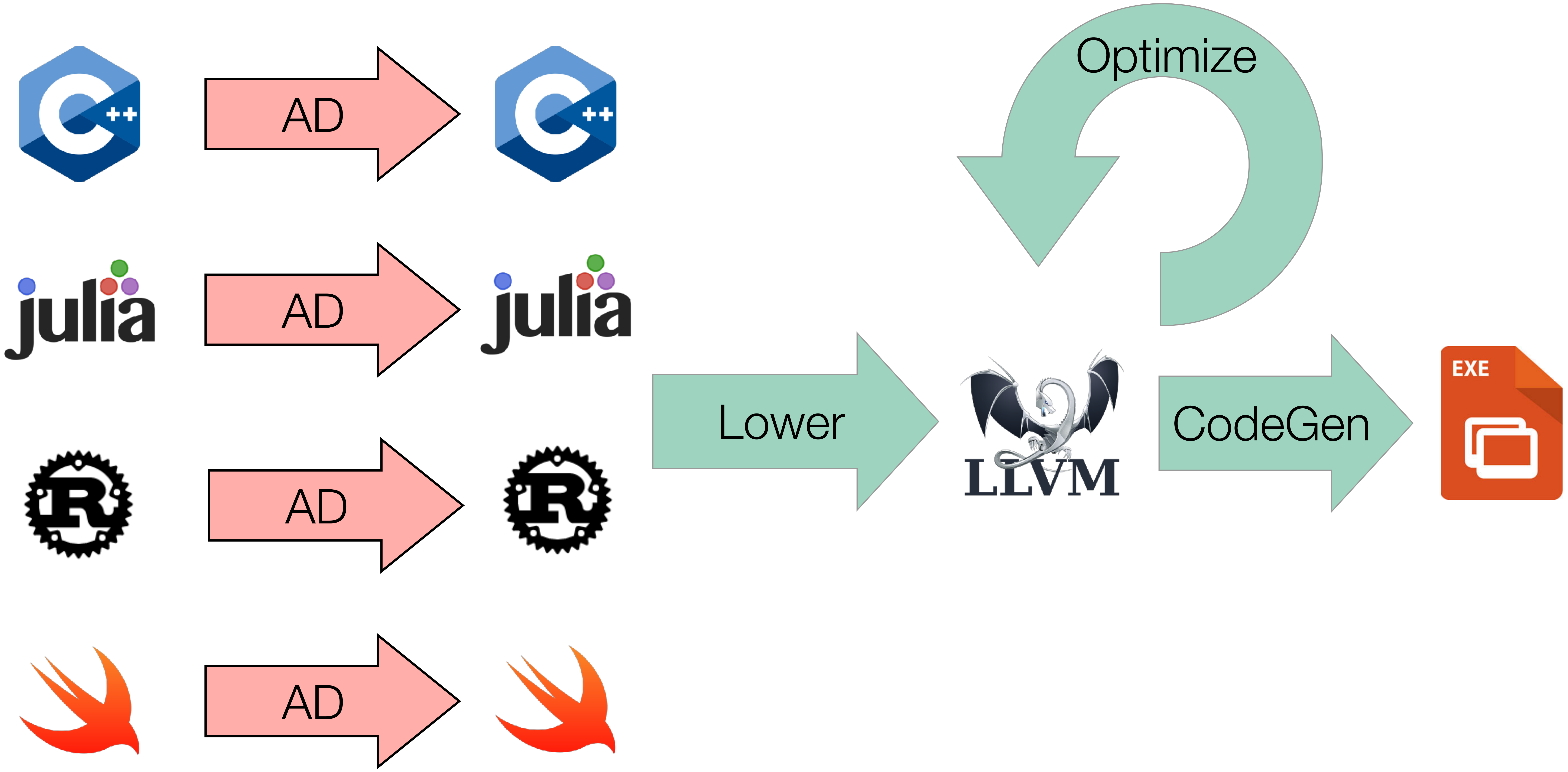
```c
// myfile.c
double relu3(double x) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```




```
// grad_myfile.h
```

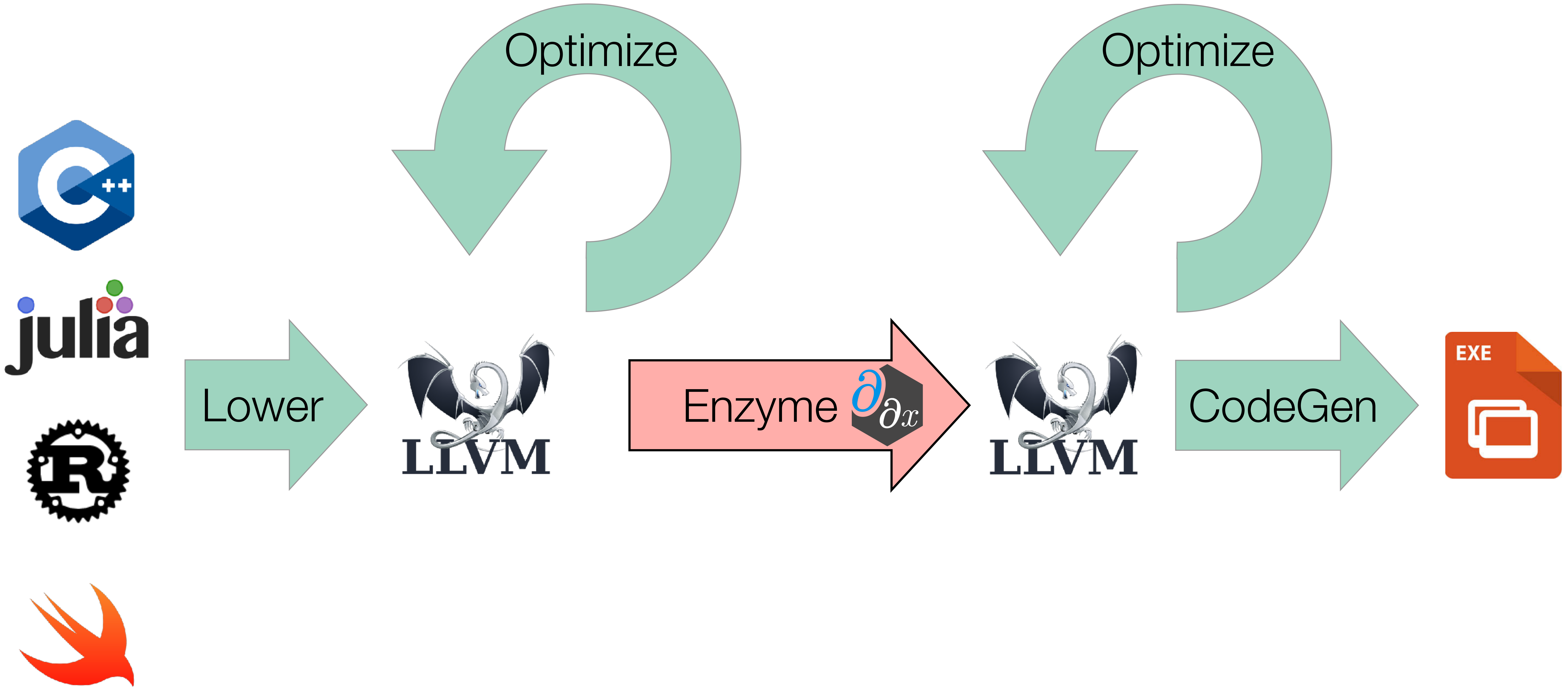
```c
// grad_myfile.c
double grad_relu3(double x) {
  if (x > 0)
    return 3 * pow(x,2)
  else
    return 0;
}
```

Existing Automatic Differentiation Pipelines



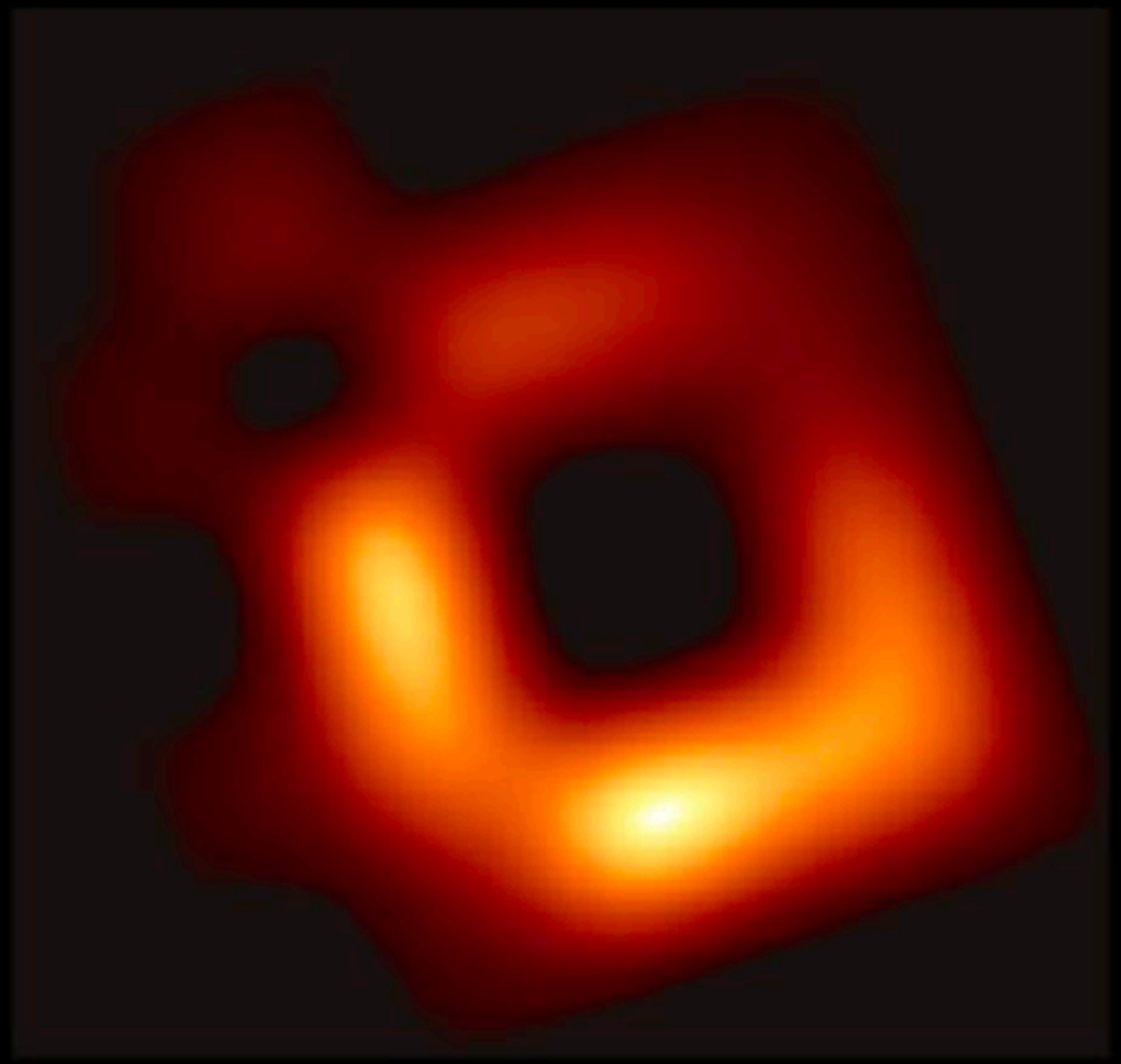
Enzyme Approach

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



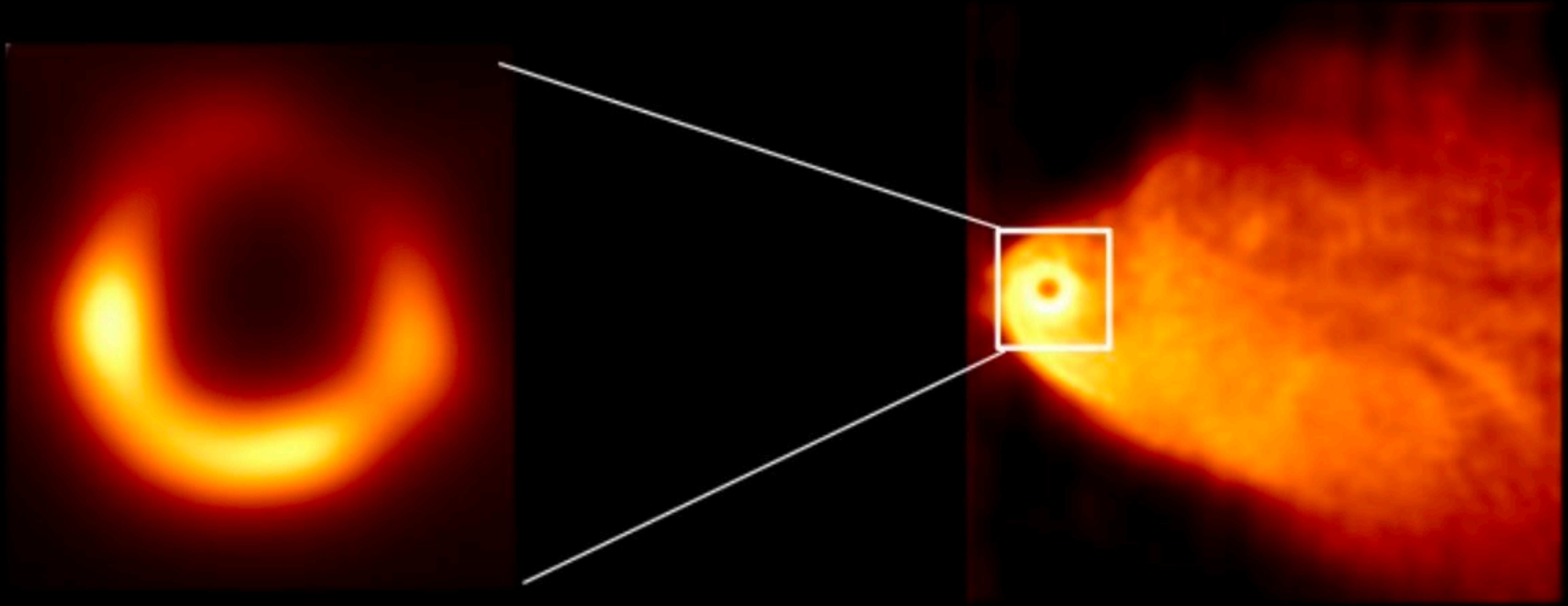
Accelerated Black Hole Imaging with Julia & Enzyme

EHT Tools M87 2017
Image Analysis: ~ **1 week** (cluster)



Julia+Enzyme M87 2017
Image Analysis: **1 hour** (1 thread)

Julia+Enzyme next-generation images
Image Analysis: **1-2 days (8 threads)**
(100x increase in computational complexity)



Simulation

Comrade.jl: Julia Bayesian Black Hole Imaging

Paul Tiede, Harvard & Smithonian | CfA

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!

$O(n^2)$

```
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

$O(n^2)$

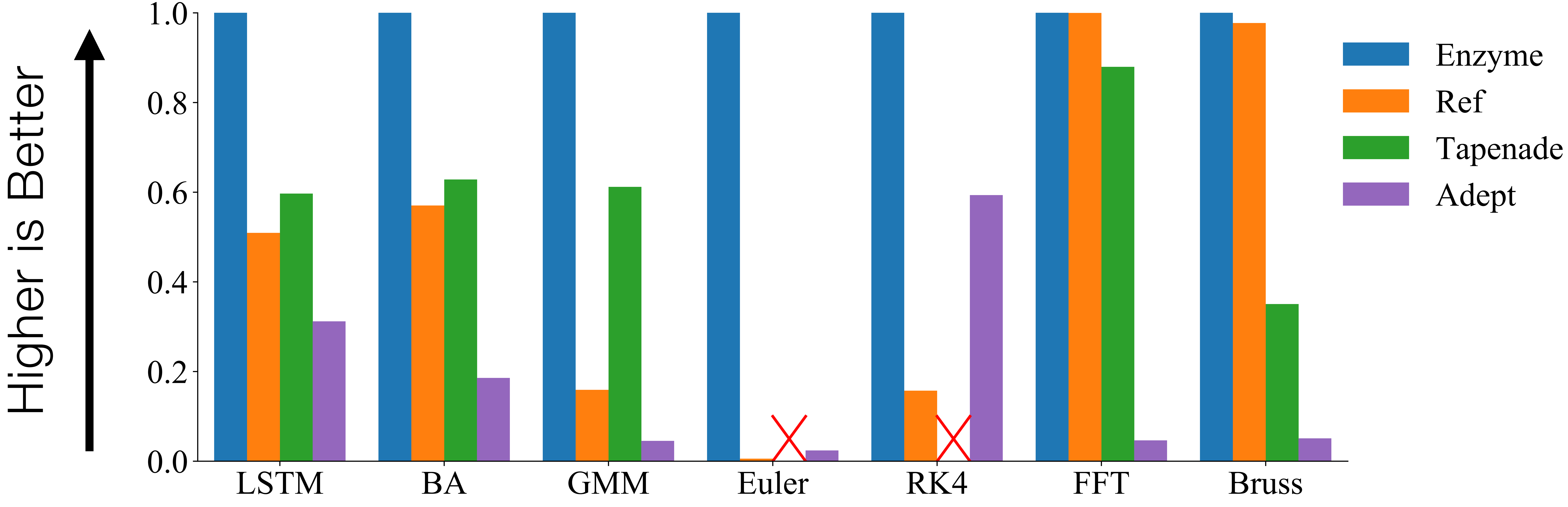
```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimize

$O(n^2)$

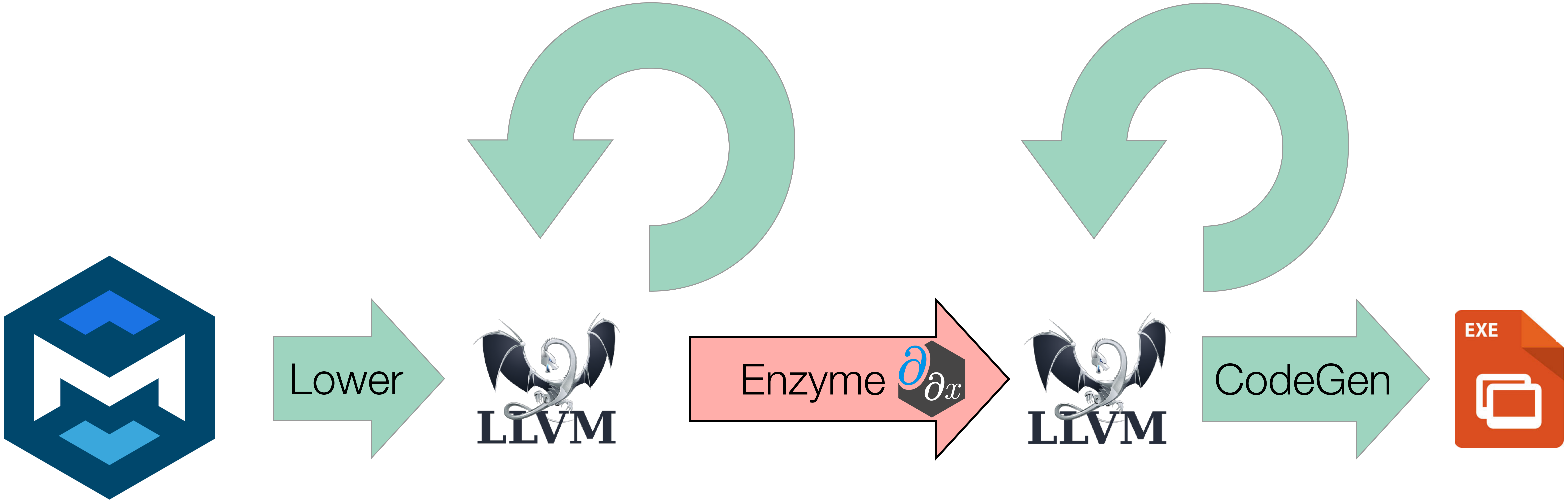
```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Enzyme CPU Speedups [NeurIPS'20]



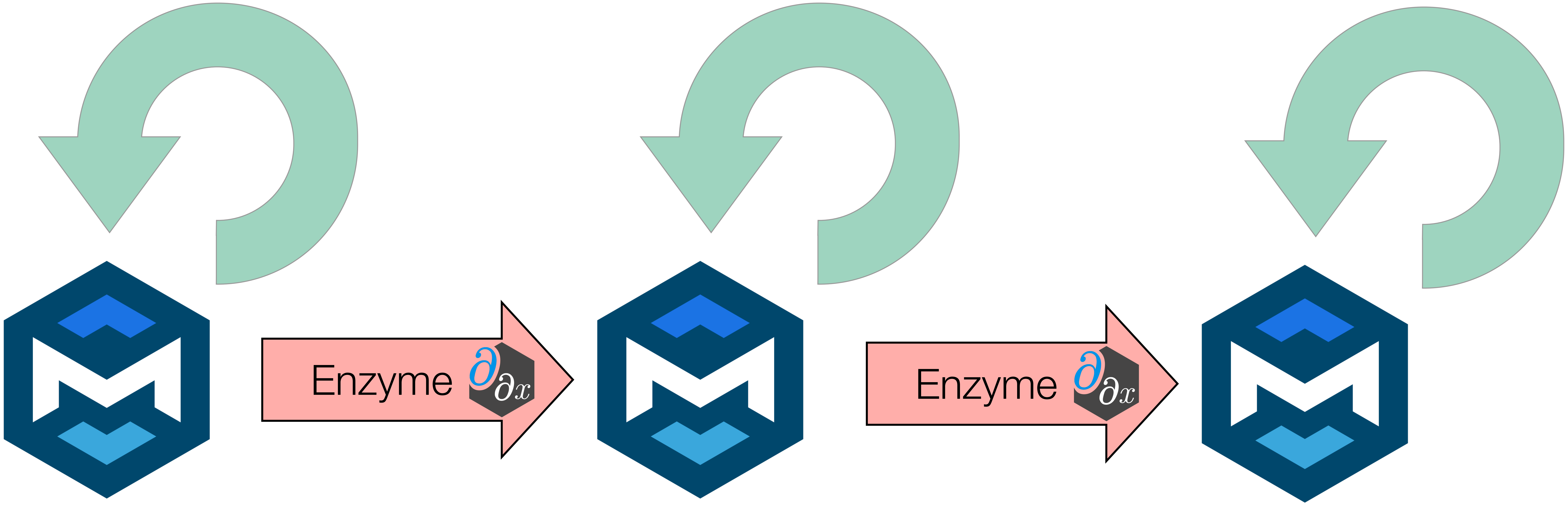
Enzyme is ***4.2x faster*** than Reference!

Why MLIR?



Why MLIR?

“Multi-level” coordination of AD and Optimization!

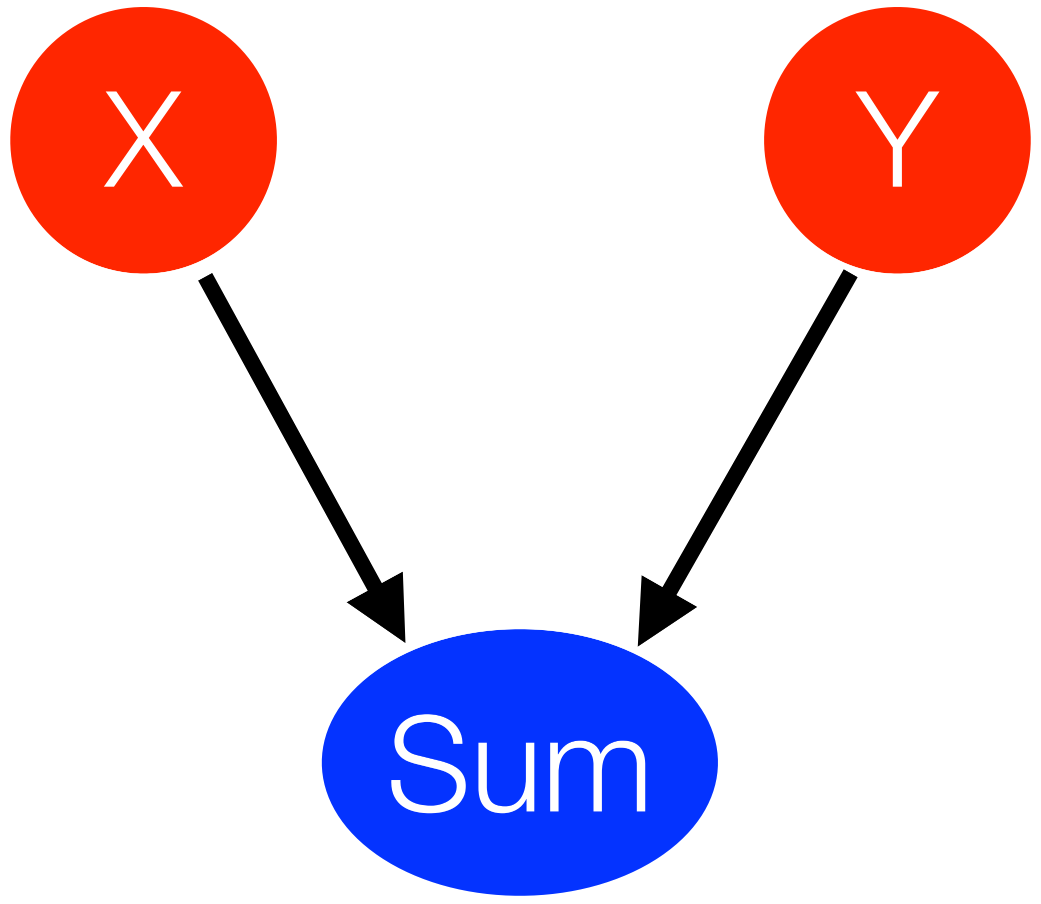


Cache Reduction [from SC'21]

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

Overwritten:

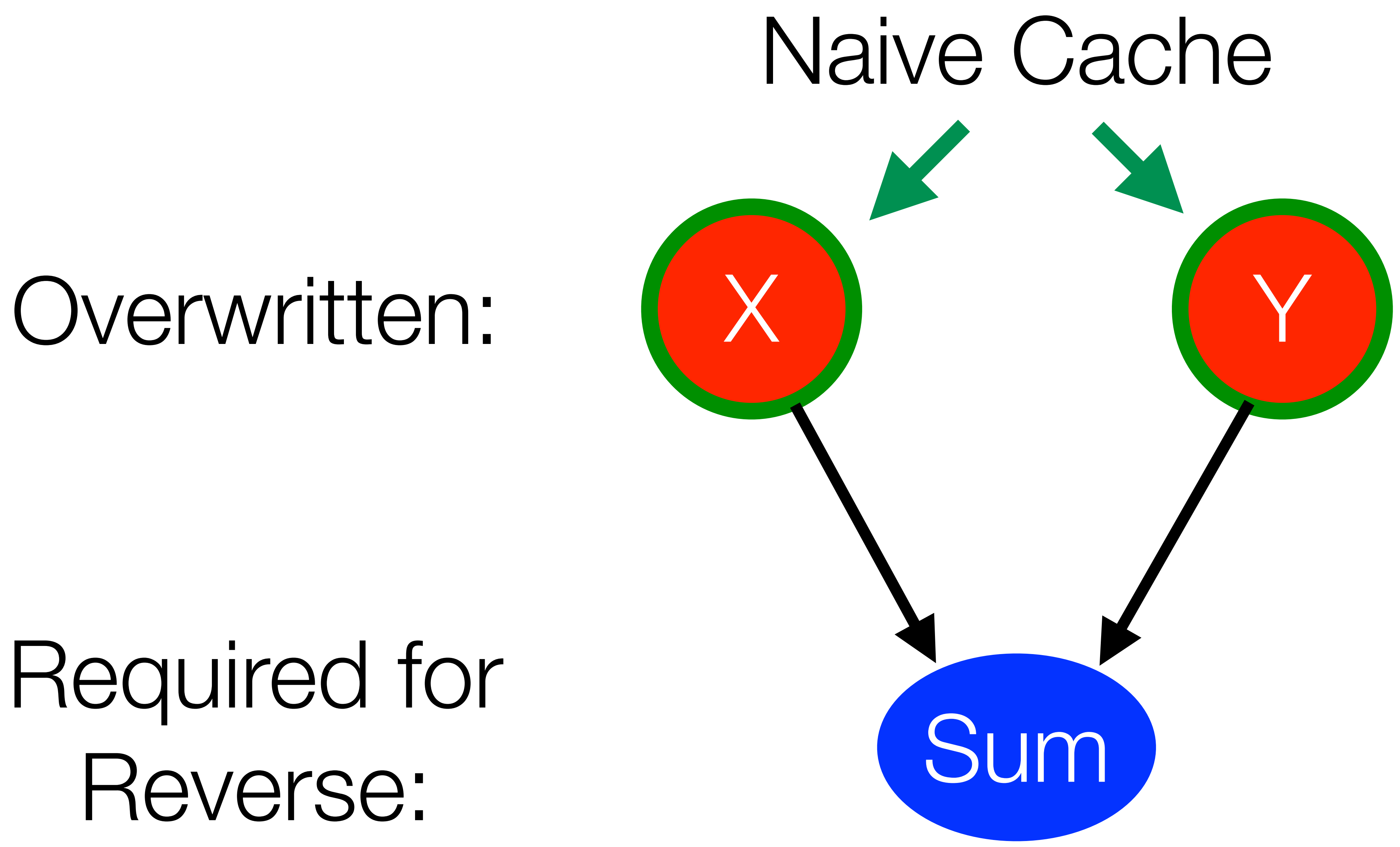
Required for Reverse:




```
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 [from SC'21]

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

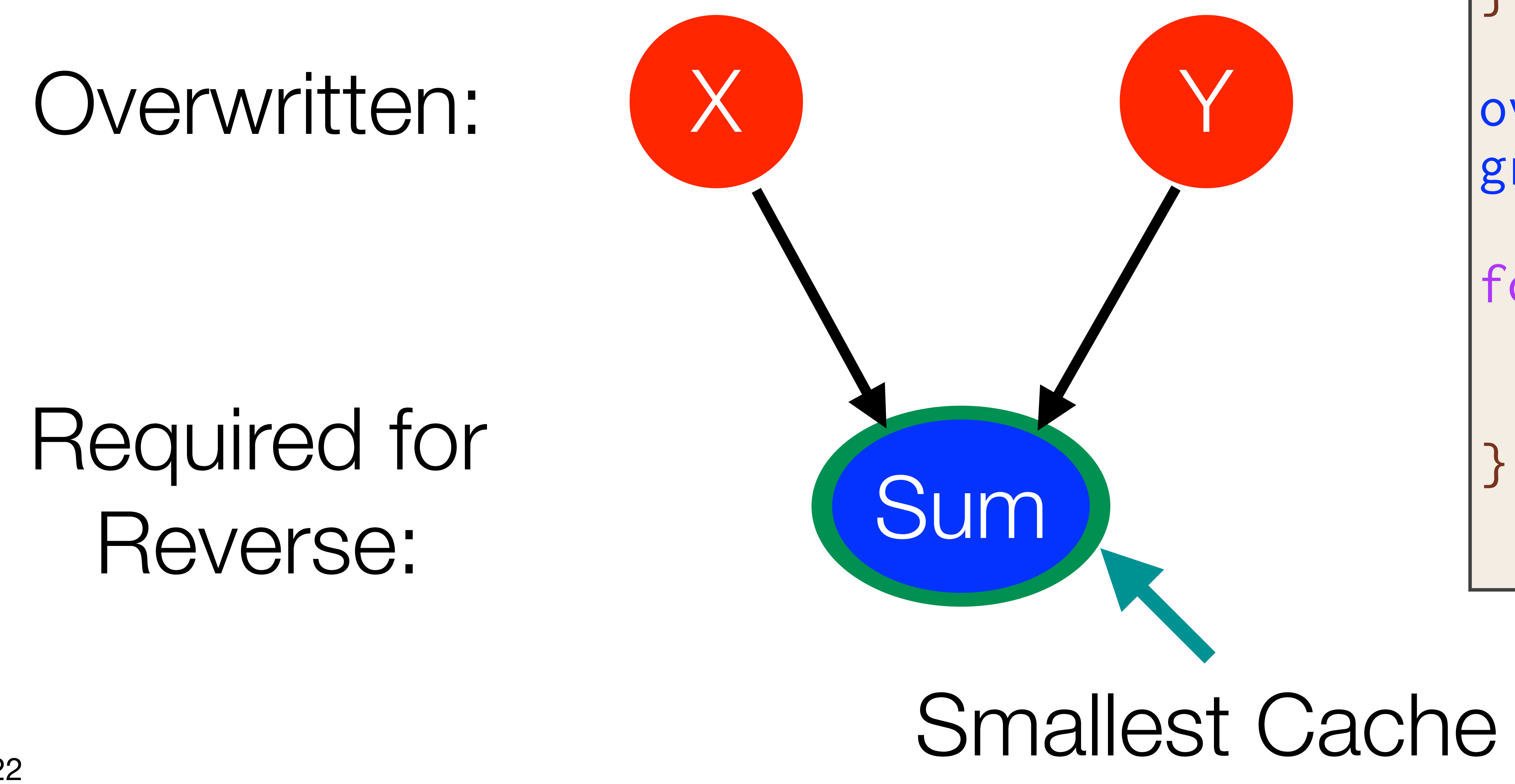



```
double* x_cache = new double[10];
double* y_cache = new double[10];

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 [from SC'21]

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

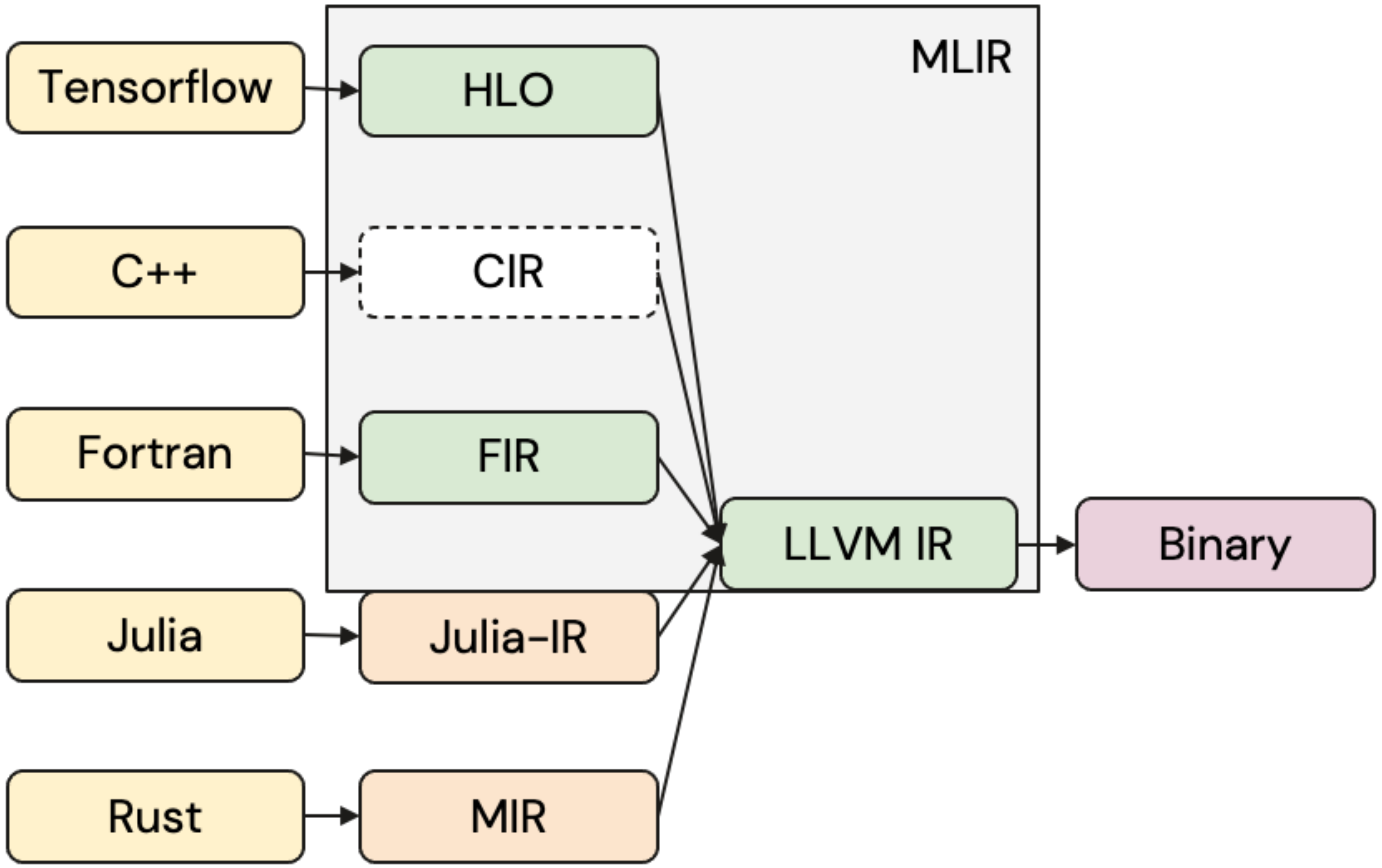
```
double* sum_cache = new double[10];

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]);
}
```



GPU Speedups [SC'21]

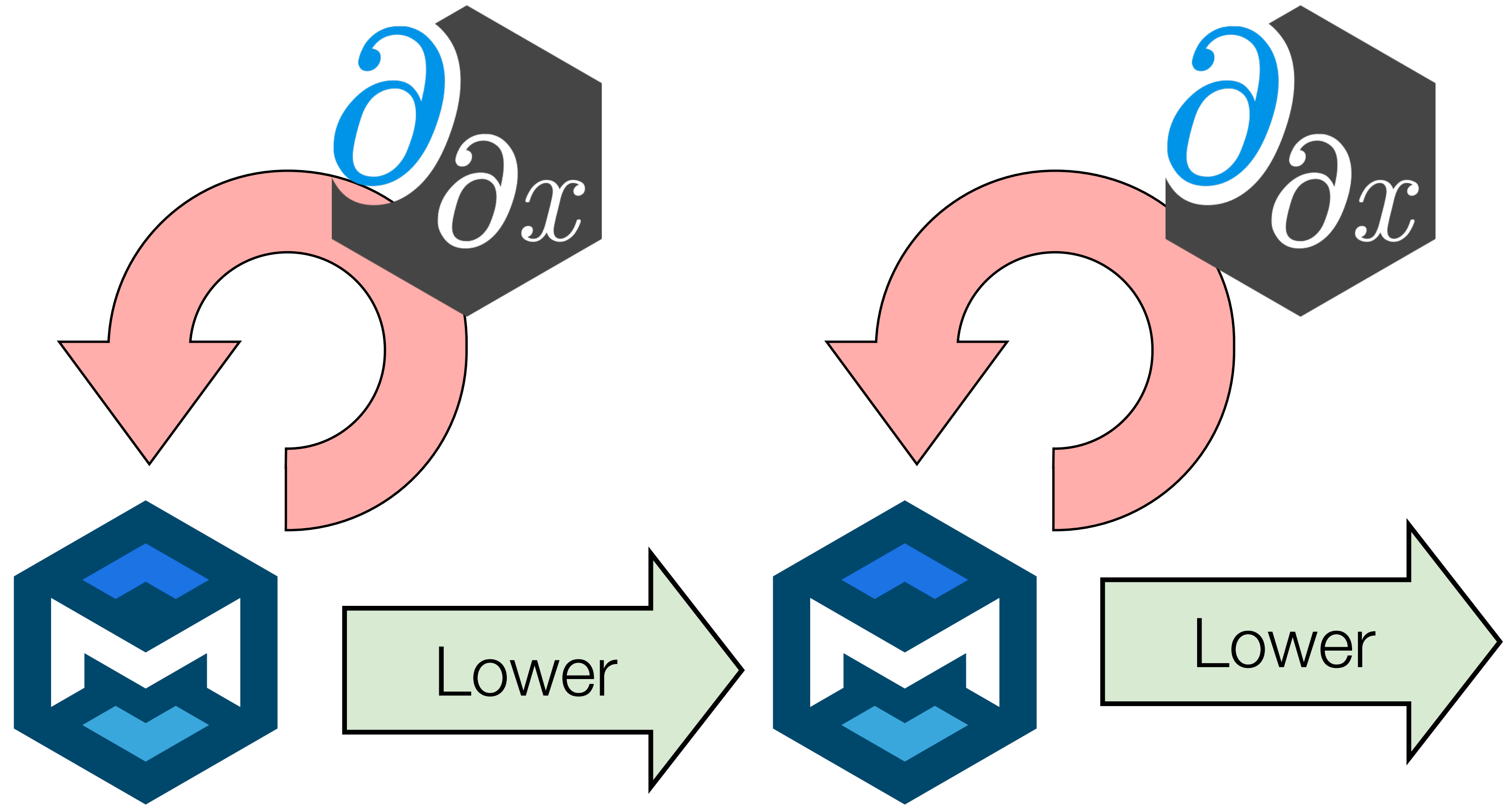
Multi-Level Differentiation

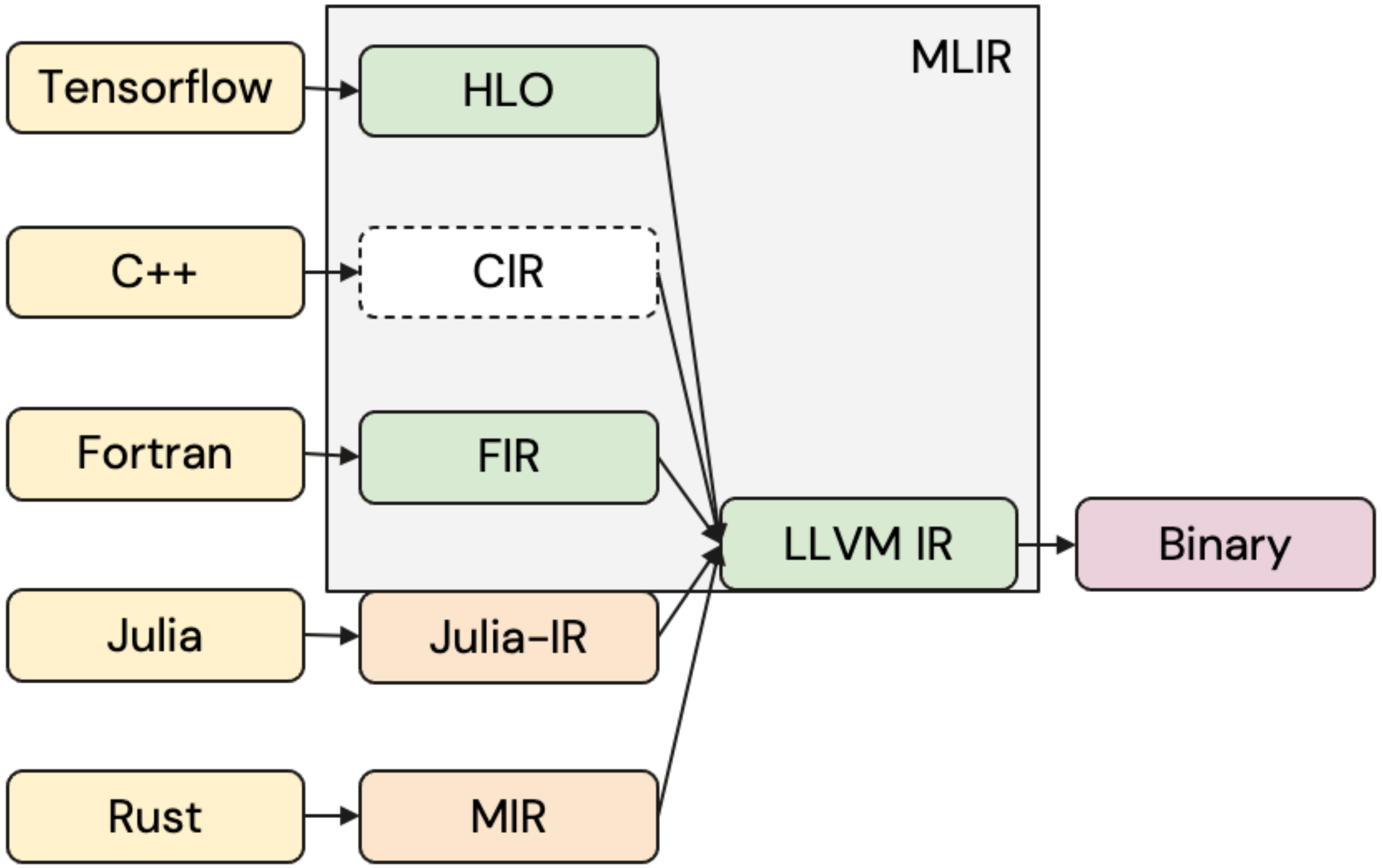
- Zoo of different MLIR dialects for various domains and optimizations
- Do we really have to write differentiation for each of the sub-abstractions again?



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





Integrating a Dialect with Enzyme

+

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

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)
```

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<f32>
```

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

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

General MLIR AD Algorithm [Reverse Mode]

Assuming one function.

```
foreach block in basic-blocks:
  foreach operation in block:
    createPlaceholderShadowValues(operation) # call shadow zero

foreach block in reverse(topological-sort(basic-blocks)):
  foreach argument in block-arguments(block):
    // uses pre-existing BranchOpInterface
    foreach source in potential-predecessor-sources(argument)
      argument.type.createAddToOp(shadow(argument), shadow(source))

    foreach operation in reverse(block):
      operation.cacheValues()
      operation.createReverseModeTangent()

    // currently hardcoded, requires generative counterpart to BranchOpInterface
    create-switch-to-successors()
```

Ongoing Work

x

Improved and Inter-Procedural Differentiation Analyses

Classical Pointer Analyses

Points-to analysis: which data may a value may point to. Aliasing analysis: whether two pointer-like values may be pointing to the same memory location.

Enzyme Type Analysis

The underlying type of a value, necessary to compute its derivative, and can also prove an operation inactive.

AD Activity Analysis

Whether a value or an operation is needed to produce a partial derivative of the given input wrt the given outputs.

Dataflow Analyses

Can be combined in one sweep to reduce overall cost.
Mutually reinforce with range, liveness, etc. analyses.

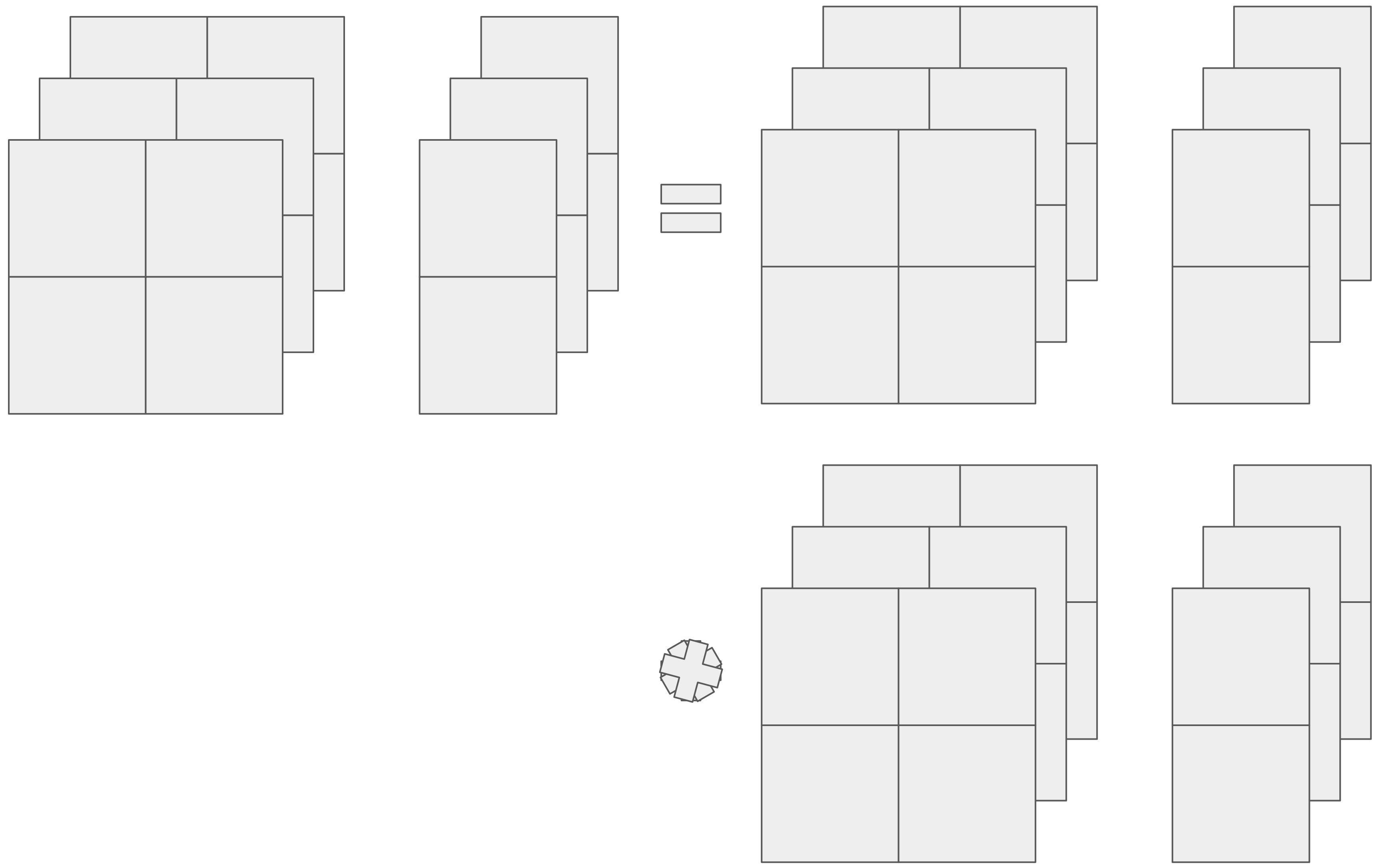
Combining Differentiation With Program Scheduling

Example: Linear Algebra Elementwise Ops

An operation extends a scalar arithmetic operation to an arbitrary-dimensional object elementwise:

```
linalg.elemwise_binary %a, %b {
  yield sqrt(%a*%a + %b*%b) : f32
} : tensor<42x10x16x17xf32>
```

- Implicit and easily reversible loop nest.
- Tape footprint computable upfront.
- Can be used to “fuse” computations and avoid caching temporaries, or “fission” them with rematerialization.



Enzyme-MLIR

- Tool for performing forward and reverse-mode AD of statically analyzable MLIR (and LLVM)
- Combining AD with optimization amplifies the impact of the optimizations!
- Multi-level (deferred) differentiation and simple interfaces enable easy integration into dialects (worst case fall back to Enzyme-LLVM)
- Ongoing work for improving analyses and combining with scheduling
- Lots of open questions (what level should each op be differentiated, how to fuse, etc)
- Open source (enzyme.mit.edu & join our mailing list)!
- Weekly open design meetings.

A Growing Enzyme Community (EnzymeCon 2023)

- 40 attendees spanning developers, users, and everywhere in between.

- 17 great talks from AD internals, to algorithms, to climate science, to physics, and beyond (https://enzyme.mit.edu/conference).

- Talks live streamed to YouTube (to be split individually):
  - Day 1 Link
  - Day 2 Link

Enzyme-MLIR

- Tool for performing forward and reverse-mode AD of statically analyzable MLIR (and LLVM)
- Combining AD with optimization amplifies the impact of the optimizations!
- Multi-level (deferred) differentiation and simple interfaces enable easy integration into dialects (worst case fall back to Enzyme-LLVM)
- Ongoing work for improving analyses and combining with scheduling
- Lots of open questions (what level should each op be differentiated, how to fuse, etc)
- Open source (enzyme.mit.edu & join our mailing list)!
- Weekly open design meetings.