Differentiable Programming in C++

VASSIL VASSILEV, WILLIAM MOSES

Speakers

Vassil Vassilev,
Research Software Engineer,
Princeton/CERN

William Moses,
Ph.D. Candidate, MIT

What is this talk about?

$$f(x)$$

Outline

- A warmup
  - Measuring the rate of change
- Introduction
  - Computing derivatives. Approaches
  - A gentle introduction to AD. Chain rule
  - Applications using AD
- Differentiable Programming
  - Deep learning & AD
  - Backpropagation
  - Existing tools & Frameworks
- Implementation
  - Discuss possible implementation approaches
  - Showcase tools built as part of the Clang/LLVM compiler toolchain.
  - Explain how such tools work and what are the benefits
- Briefly outline standardization efforts (as per https://wg21.link/P2072)
- Conclusion

How fast he ran?

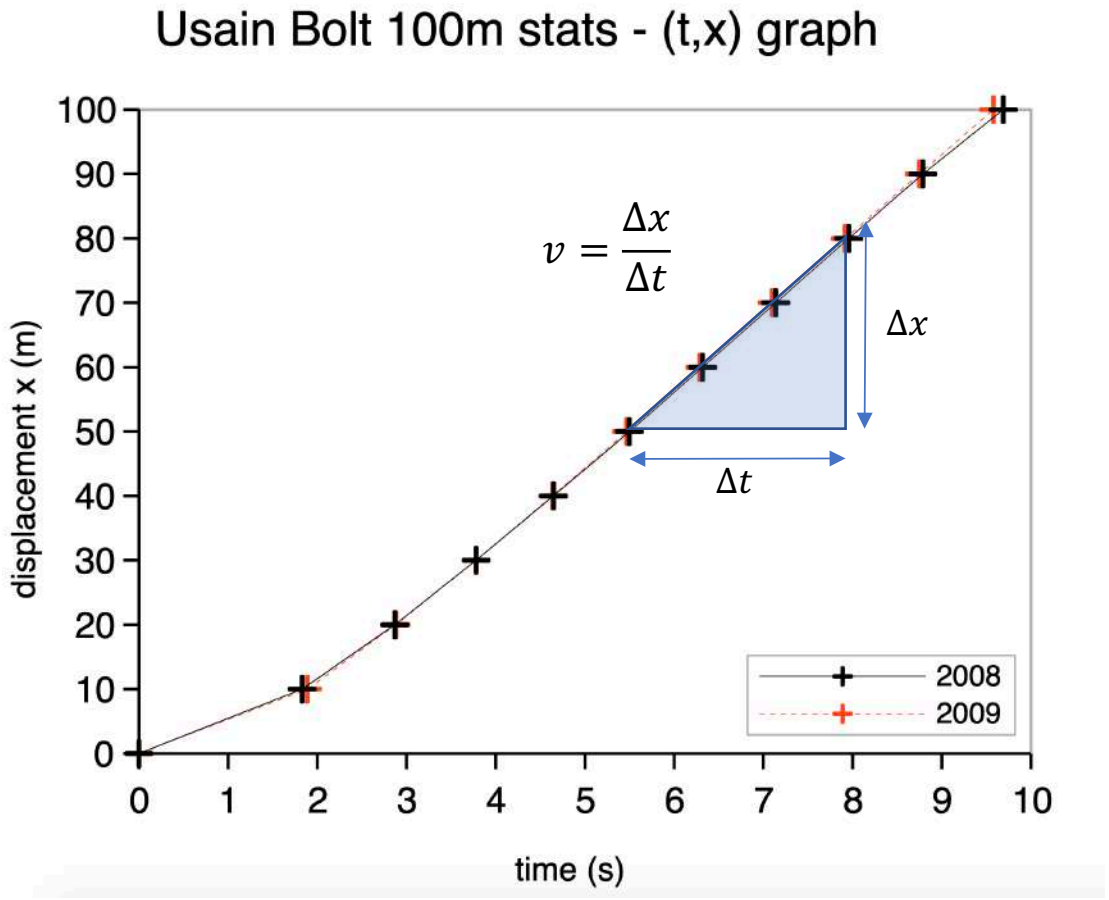
How fast he ran? What does that even mean?

Displacement = velocity * time

100/9.58 = 10.44 m/s => 37.58 km/h on average

- Did he accelerate until the end?
- When did he slow down?
- What was his top speed?

Measuring the rate of change



Plot credits: A. Penev

Data from SportEndurance.com

Bolt (m)	2008 (s)	2009 (s)
0	0	0
10	1.83	1.89
20	2.87	2.88
30	3.78	3.78
40	4.65	4.64
50	5.5	5.47
60	6.32	6.29
70	7.14	7.1
80	7.96	7.92
90	8.79	8.75
100	9.69	9.58

To find the time and velocity at some interval we could calculate the *gradient* graph at *different* times.

For example the velocity of Bolt from the 50th to the 80th meter was:

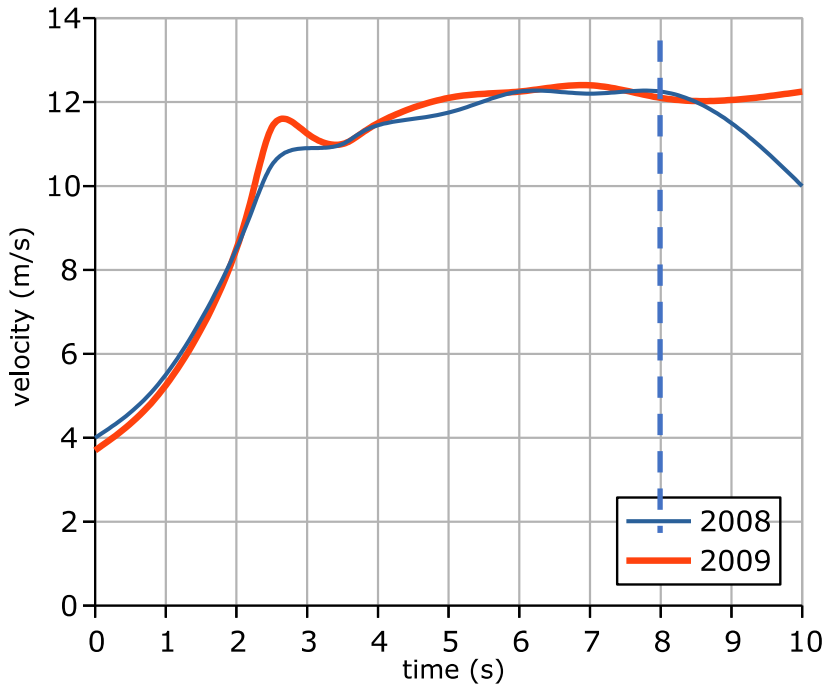
$$v = \frac{\Delta x}{\Delta t} = \frac{80 - 50}{7.96 - 5.5} = 12.19 m/s$$

Could he do better in 2009?

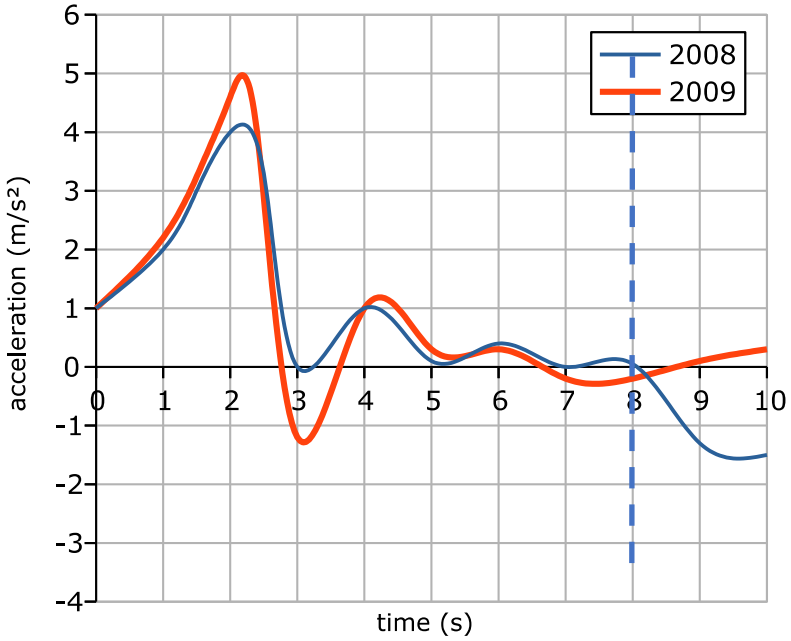
Bolt, 100m dash, Beijing Olympics, 2008, source [quantamagazine.org]

Derivatives: measure the rate of change

velocity



acceleration

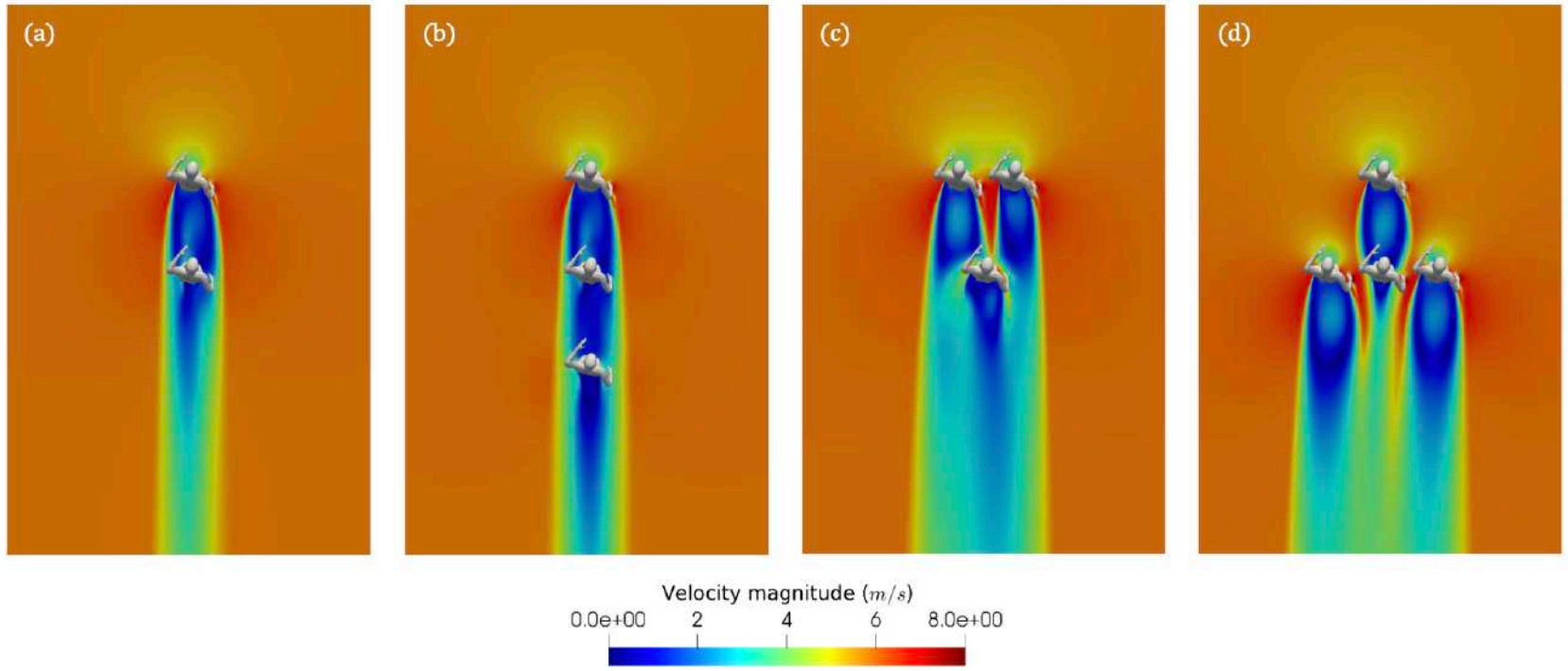


A derivative measures the rate of a function's output value wrt a change in its input:

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

Plot credits: A. Penev

The longer the distance the more parameters



Schickhofer, Lukas, and Henry Hanson. "Aerodynamic effects and performance improvements of running in drafting formations." *Journal of Biomechanics* 122 (2021): 110457.

Tactics are skills required in a competition that allow a player or team to effectively use their talent and skill to the best possible advantage. Usually means to empirically develop an intuition how to win and apply it.

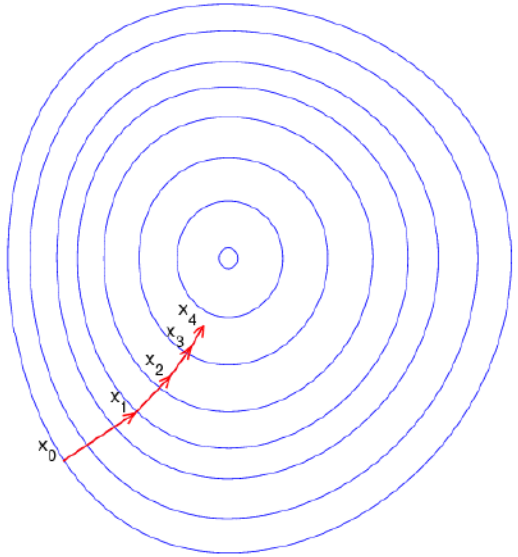
Building a reference trajectory with a goal of maximizing performance (output) while minimizing the set of inputs.

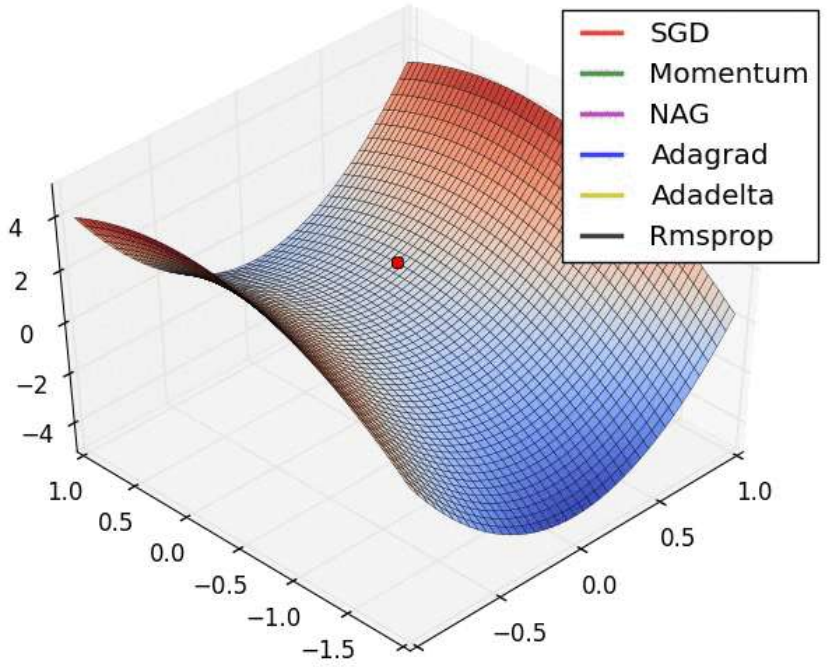
Thus, we need to know how each input parameter affects the output.

Gradient Descent

A gradient is the vector of values of the function; each entry is the output of the function's derivative wrt a parameter…

$$\nabla f(x1, \ldots, x_n) = \begin{bmatrix} \frac{\partial f}{x_1}(x1, \ldots, x_n) \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial f}{x_n}(x1, \ldots, x_n) \end{bmatrix}$$





Plot credits: https://ruder.io/optimizing-gradient-descent/

The gradient vector can be interpreted as the "direction and rate of fastest increase"

Computing Derivatives

Computing Derivatives

Manual

- Error prone

Numerical Differentiation (ND)

- Precision errors
- High computational complexity
- Higher order problem (formula approximated by missing higher order terms)

Symbolic Differentiation (SD)

- Only works on single mathematical expressions (no control flow)
- May require transcribing result back into code

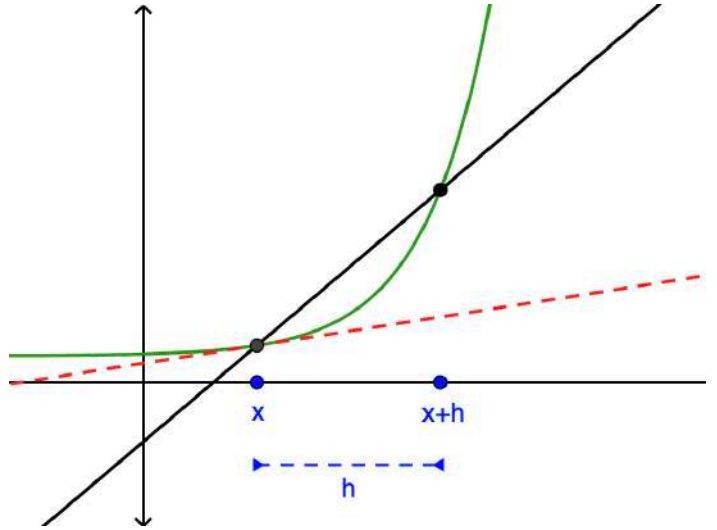
Algorithmic or Automatic Differentiation (AD)

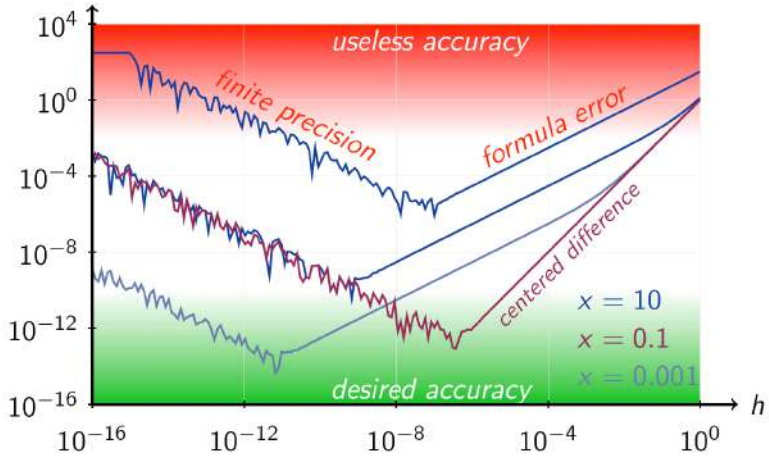
- Automatically generate a C++ program to compute the derivative of a given function

Numerical Differentiation

$$\frac{df(x)}{dx} \approx \frac{f(x) - f(x+h)}{h}$$

- The choice of *h* is problem-dependent.
- Too big step *h* makes the approximation too poor
- Too small *h* makes the floating point round-off error too big
- The computational complexity is O(n), where n is the number of parameters – for a function with 100 parameters we need 101 evaluations





Symbolic Differentiation

- Limited to closed form expressions
- Requires a symbolic processing system (eg Mathematica, Mapple) and transcribing back the algorithm
- Suffers from expression swell (subexpression accumulation), especially challenging when going to higher order derivatives

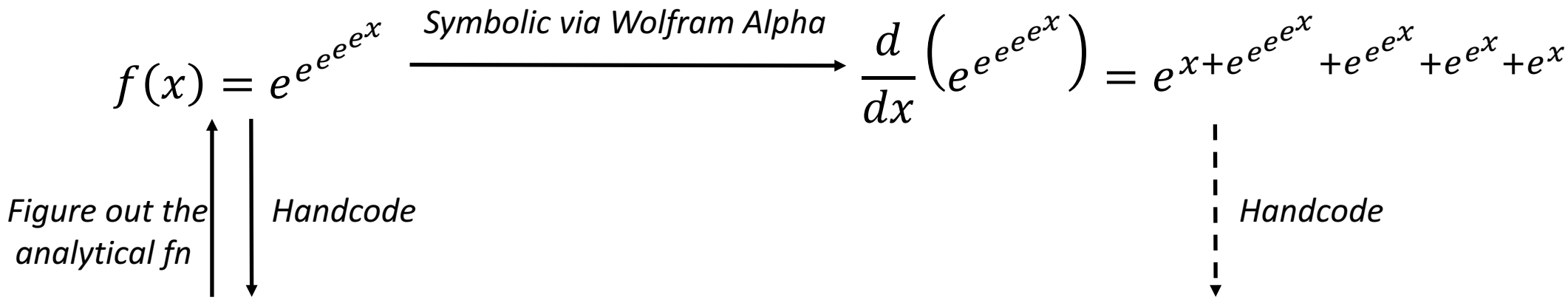
```
// Supports
double pow3(double x) {
  return x * x * x;
}
// Does not support
double pow3_(double x) {
  if (x == 0) return 0;
  return x * x * x;
}
```

Automatic Differentiation

”[AD] is a set of techniques to evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.).” [Wikipedia]

Known as algorithmic differentiation, autodiff, algodiff, computational differentiation.

Automatic and Symbolic Differentiation




```
// f(x)=e^(e^(e^(e^(e^x))))
#include <cmath>
double f (double x) {
  double result = x;
  for (unsigned i = 0; i < 5; i++)
    result = std::exp(result);
  return result;
}
```

AD

```
double f_dx(double x) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < 5; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

AD. Chain Rule

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

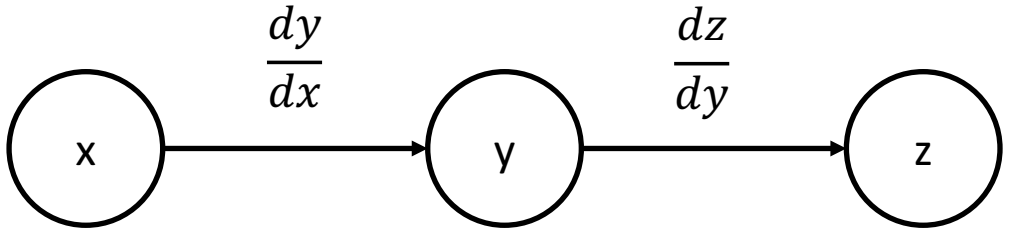
Intuitively, the chain rule states that knowing the instantaneous rate of change of *z* relative to *y* and that of *y* relative to *x* allows one to calculate the instantaneous rate of change of *z* relative to *x* as the product of the two rates of change.

“if a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels 2 × 4 = 8 times as fast as the man.” G. Simmons

AD. Algorithm Decomposition

```
y = f(x)
z = g(y)
```

```
dydx = dfdx(x)
dzdy = dgdy(y)
dzdx = dzdy * dydx
```

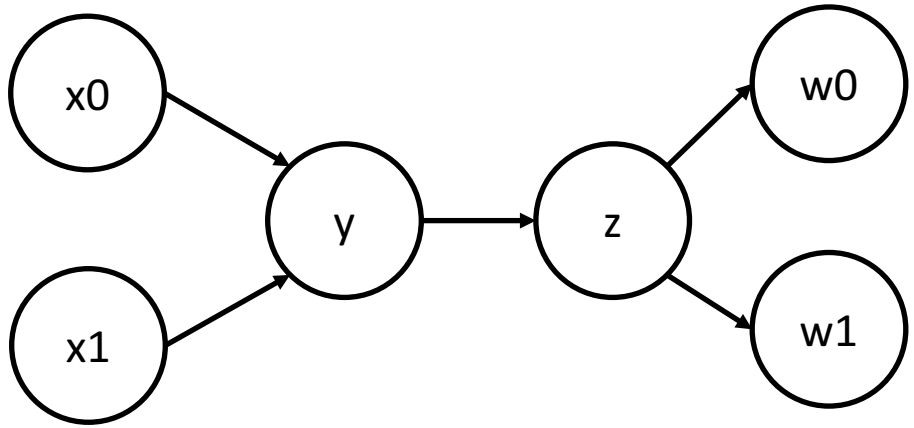


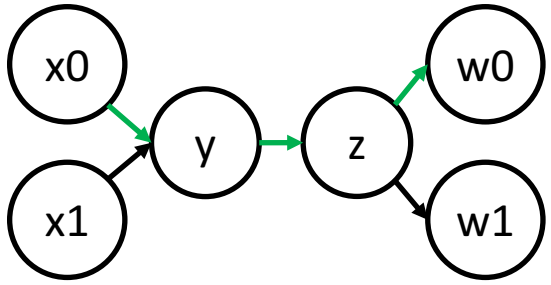
In the computational graph each node is a variable and each edge is derivatives between adjacent edges

We recursively apply the rules until we encounter an elementary function such as addition, multiplication, division, sin, cos or exp.

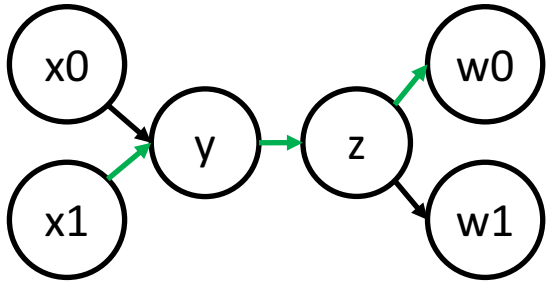
AD. Chain Rule

```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)
```

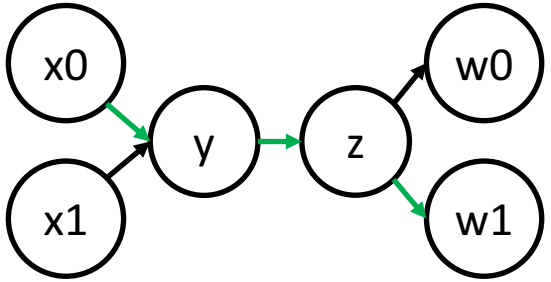




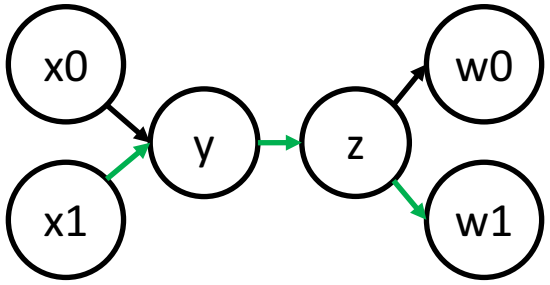
$$\frac{\partial w0}{\partial x0} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w0}{\partial x1} = \frac{\partial w0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$



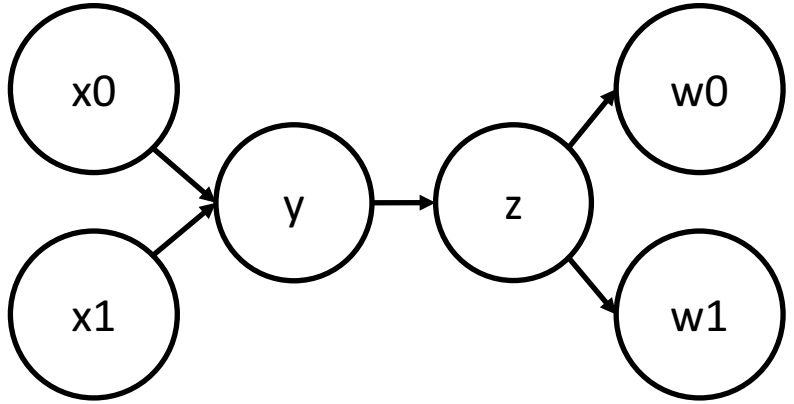
$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$



$$\frac{\partial w1}{\partial x1} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x1}$$

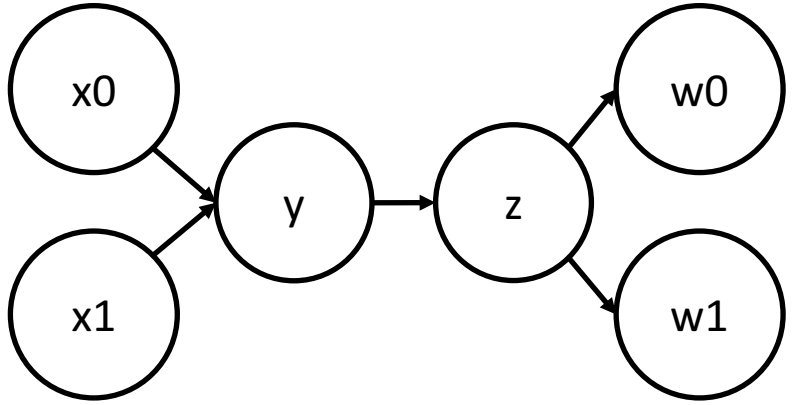
AD step-by-step. Forward Mode

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx, x1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = l(z)
dw0dx, dw1dx = dl(z, dzdx)
```



AD step-by-step. Reverse Mode

```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}

dwdz = dl(dwdw0, dwdw1)

dwdy = dg(y, dwdz)

dwx0, dwx1 = df(x0, x1, dwdy)
```



AD Control Flow

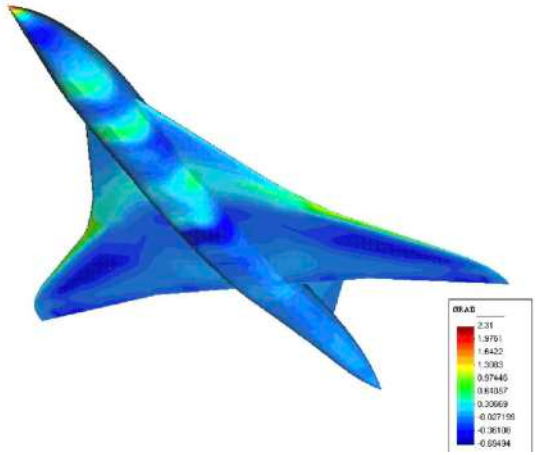
- Control Flow and Recursion fall naturally in forward mode.
- Extra work is required for reverse mode in reverting the loop and storing the intermediaries.

```
double f_reverse (double x) {
  double result = x;
  std::stack<double> results;
  for (unsigned i = 0; i < 5; i++) {
    results.push(result);
    result = std::exp(result);
  }
  double d_result = 1;
  for (unsigned i = 5; i; i--) {
    d_result *= std::exp(results.top());
    results.pop();
  }
  return d_result;
}
```

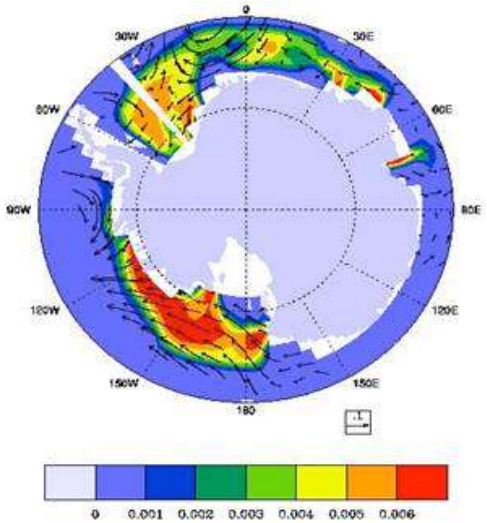
AD. Cheap Gradient Principle

- The computational graph has **common subpaths** which can be precomputed
- If a function has a single input parameter, no mater how many output parameters, **forward mode** AD generates a **derivative** that has the **same time complexity** as the original function
- More importantly, if a function has a **single output** parameter, **no matter how many input** parameters, reverse mode AD generates **derivative** with the **same time complexity** as the original function.

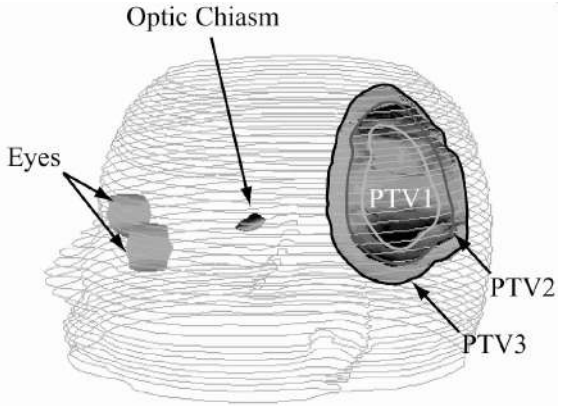
Uses of AD outside of Deep Learning



Gradient of the Sonic Boom objective function on the skin of the plane, CFD, Laurent Hascoët et al.



Sensitivities of a Global
Sea-Ice Model, Climate, Jong G. Kim et al



Intensity Modulated Radiation Therapy, Biomedicine, Kyung-Wook Jee et al

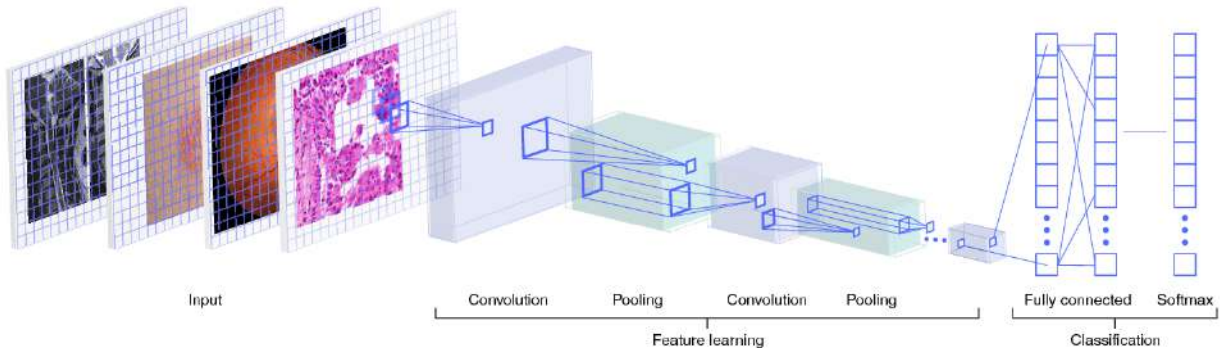
Differentiable Programming

Deep Learning & Automatic Differentiation

Imagined by GAN, ThisPersonDoesNotExist.com

Image colorization

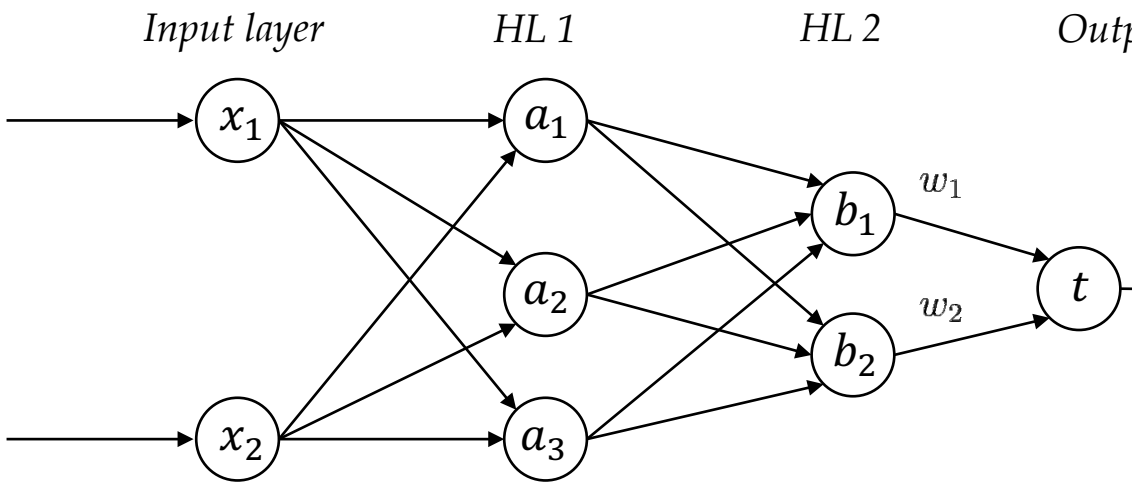
Tesla Autopilot, tesla.com



Medical Imaging, CNN, A. Esteva et al, A guide to deep learning in healthcare

Speech Recognition

Backpropagation



$$y = \phi\left(\sum_i w_i x_i\right)$$

$$E = \frac{1}{2}(t - y)^2$$

Error at output, the error between observed and desired state. Computed from the output *y* and seen desired output *t*.

Training pattern is fed, forward generating corresponding output

Forward pass – make a prediction

Calculate Loss

$$\Delta w_i = \eta \frac{\partial E}{\partial w_i}$$

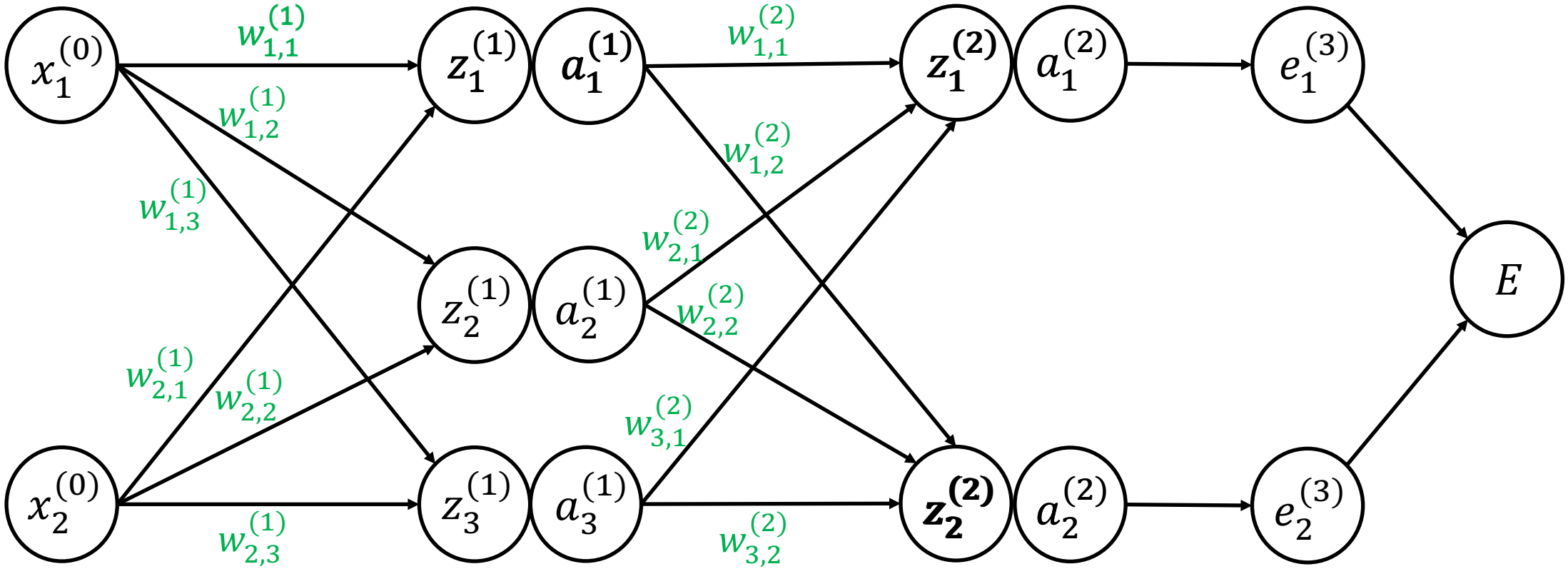
Backpropagation – adjust the weights to minimize loss

The error propagates back, through updates of the subtracted gradient ratio from the weights.

x_i, w_i, ϕ, η *are inputs, input weights, activation function and learning rate of the neuron*

Backpropagation

$$\frac{\partial}{\partial} = \left(\frac{\partial}{\partial} \frac{\partial}{\partial} \frac{\partial}{\partial} + \frac{\partial}{\partial} \frac{\partial}{\partial} \frac{\partial}{\partial} \right) \frac{\partial}{\partial} \frac{\partial}{\partial}$$



Differentiable Programming

"A programming paradigm in which a numeric computer program can be differentiated throughout via **automatic differentiation**. This allows for gradient based optimization of parameters in the program, often via gradient descent." [Wikipedia]

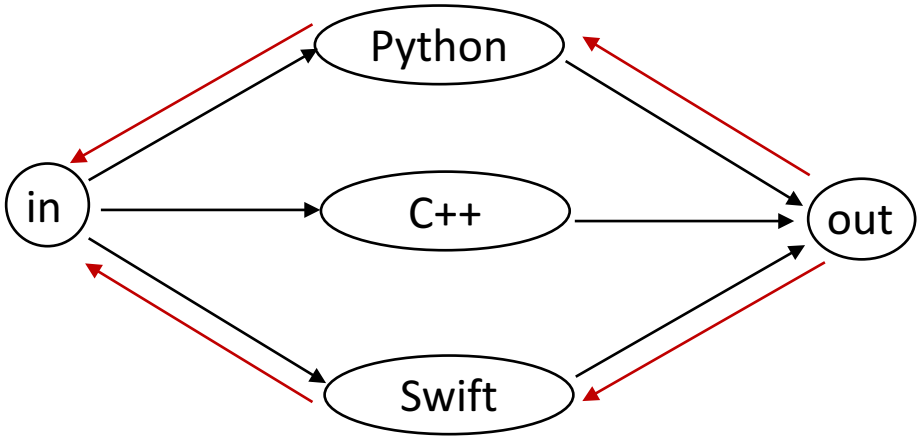
- Deep learning drives recent advancements in automatic differentiation
- AD is useful also in bayesian inference, uncertainty quantification, modeling, simulation
- Several programming languages and frameworks have enabled the differential programming paradigm by adding support for AD.
- Swift, Kotlin, and Julia have made AD a first-class citizen.

Automatic Differentiation & C++

Interoperable Machine Learning

"[The key challenge of scientific ML is that] if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train." -Rackauckas

- Limited support for C++ automatic differentiation hinders the use of C++ within machine learning
- Cannot easily use the vast set of existing C++ codebases in ML applications



C++ Automatic Differentiation Wish-List

- Fast
  - Compilation Time (ideally not JIT)
  - Execution Time
- Works on existing code
  - Doesn't require rewriting user code
  - Supports (most) C++
- Easily Maintainable
  - Minimal impact outside of AD (e.g. no rewrite of STL)
  - Keeps up with evolving standards

Existing AD Approaches (1/3)

- Differentiable DSL (TensorFlow, PyTorch, DiffTaichi, Halide)
  - 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

```
#include "tensorflow/core/public/session.h"

GraphDef graph_def;
session->Create(graph_def);
...
session->Run(inputs,{"output_class/Softmax:0"}, {}, &outputs);
```

Existing AD Approaches (2/3)

- Operator overloading (Adept [C++], JAX [Python])
  - Provide differentiable versions of existing language constructs
  - May require writing to use non-standard utilities
  - Often dynamic: storing instructions/values to later be interpreted

```
template<typename T> square(T val) { return val * val; }

adept::Stack stack;
adept::adouble inp = 3.14;
adept::adouble out(square(inp));
out.set_gradient(3.14);

double derivative = out.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 (hard for C++ & must keep up with standard)
  - Requires all code to be available ahead of time

```
double square(double val) { return val * val; }
```

```
tapenade  -b  -o out.c  -head "square(val)/(out)" square.c
```

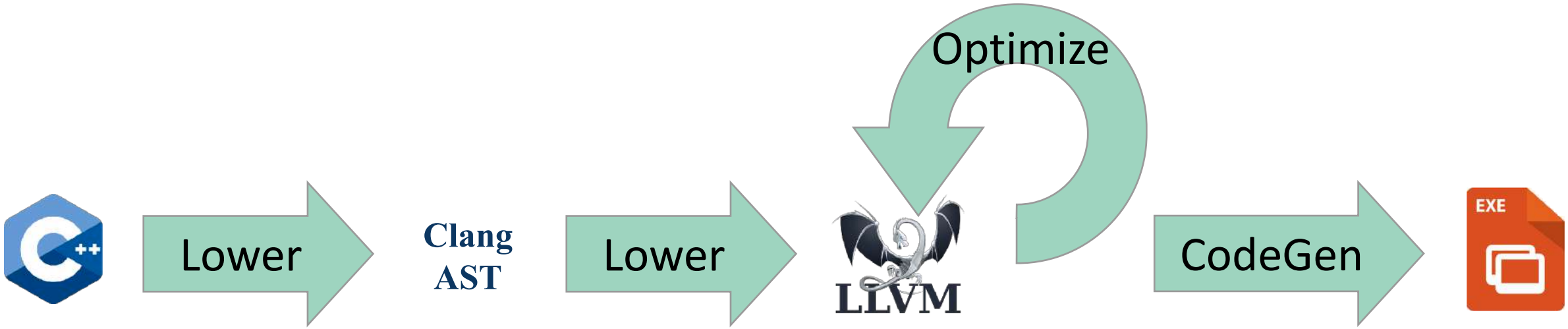
```
double grad_square(double val) { return 2 * val; }
```

Idea: Compiler-Based AD!

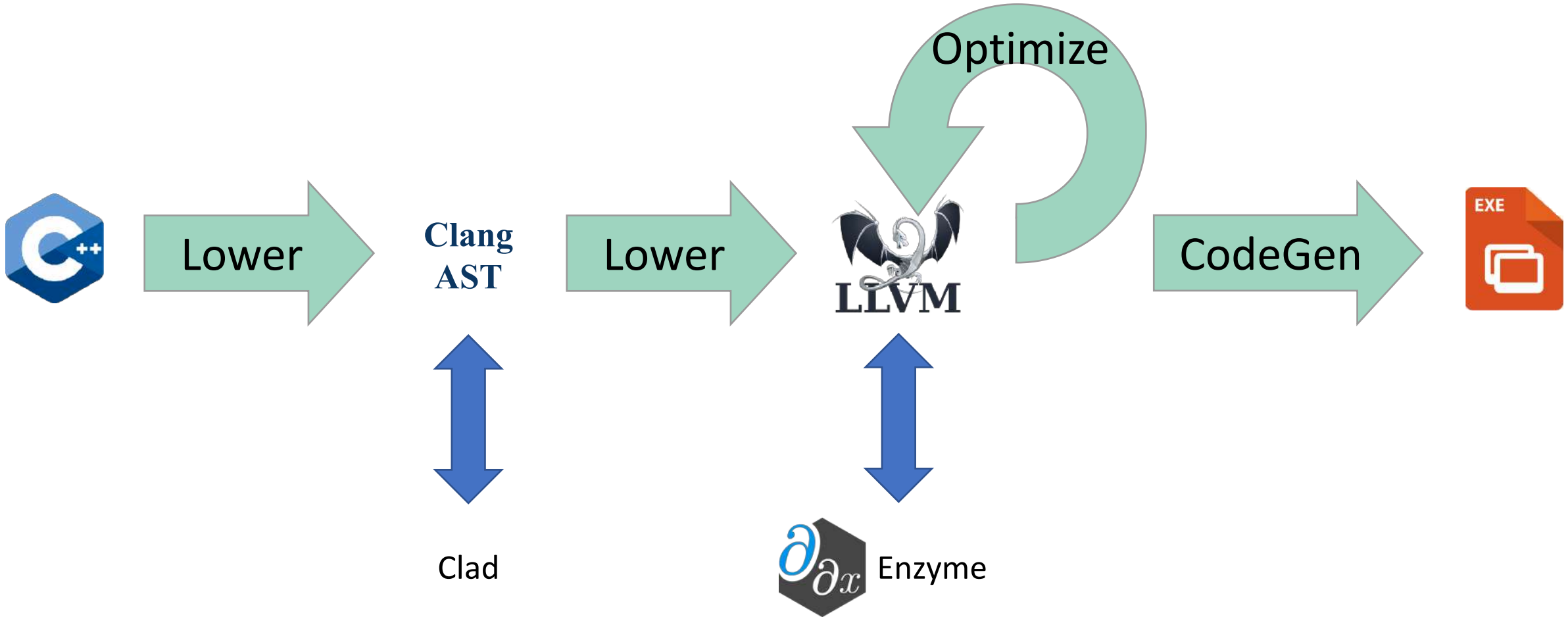
- Want the no user-rewriting, speed, and low STL-rewriting impact of source AD
- Do not want the extra maintenance burden
- Since the compiler already implements parsing, semantic analysis, etc, we can use the compiler to perform source-based AD without maintaining a second parser!

Two Case Studies of Compiler-Based AD

Implementation of AD in Clang/LLVM



Implementation of AD in Clang/LLVM



Case Study 1: Clad – AD of Clang AST

```
double square(double val) {
  return val * val;
}
```

Clang

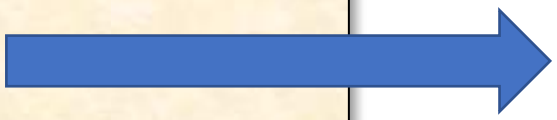
```
FunctionDecl square double (double)
|-ParmVarDecl val double
`-CompoundStmt
  `-ReturnStmt
    `-BinaryOperator double *
      |-ImplicitCastExpr double <LValueToRValue>
      | `-DeclRefExpr double ParmVar val
      `-ImplicitCastExpr double <LValueToRValue>
        `-DeclRefExpr double ParmVar val
```

Clad

```
FunctionDecl square_darg0 double (double)
|-ParmVarDecl val double
`-CompoundStmt
  |-DeclStmt
  | `-VarDecl d_val double
  |   `-ImplicitCastExpr double  <IntegralToFloating>
  |     `-IntegerLiteral int 1
  `-ReturnStmt
    `-BinaryOperator double +
      |-BinaryOperator double *
      | |-ImplicitCastExpr double  <LValueToRValue>
      | | `-DeclRefExpr double  Var d_val
      | `-ImplicitCastExpr double  <LValueToRValue>
      |   `-DeclRefExpr double  ParmVar val
      `-BinaryOperator double *
        |-ImplicitCastExpr double  <LValueToRValue>
        | `-DeclRefExpr double  ParmVar val
        `-ImplicitCastExpr double  <LValueToRValue>
          `-DeclRefExpr double  lvalue Var d_val
```

Case Study 1: Clad – AD of Clang AST

```
#include "clad/Differentiator/Differentiator.h"
double square(double val) {
  return val * val;
}

int main() {
 auto dfdx = clad::differentiate(pow2, 0);

 double res = dfdx.execute(1);

 // OR
 auto dfdxFnPtr = dfdx.getFunctionPtr();
 dfdx = dfdxFnPtr(2);

 printf("%s\n", dfdx.getCode());

 ...
}
```



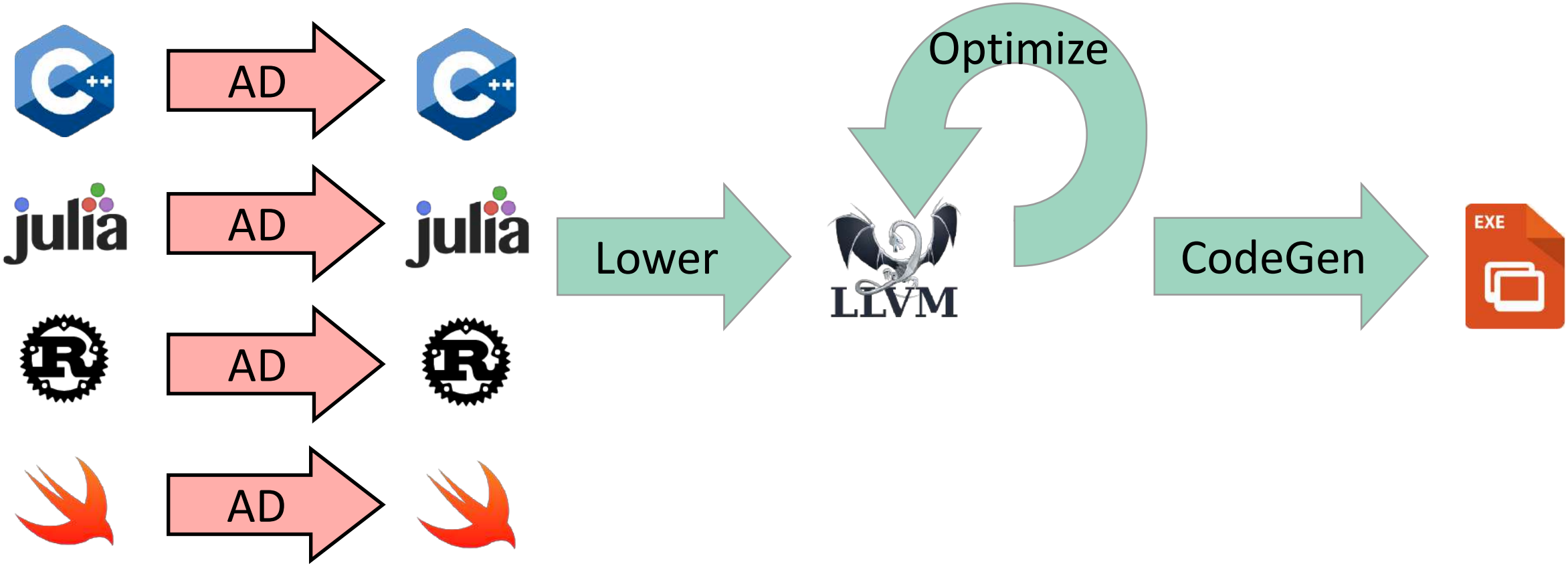
```
double square_darg0(double val) {
  double d_val = 1;
  return d_val * val + val * d_val;
}
```

Clad Key Insights

- Works on the compiler frontend level and uses the tree-rebuilding approach like the C++ template instantiator
- Can produce valid C++ source code

https://clad.readthedocs.io / https://github.com/vgvassilev/clad

Existing Automatic Differentiation Pipelines



Vector Normalization

$O(n^2)$

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

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

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

Vector Normalization: LICM

$O(n)$

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

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

Vector Normalization: LICM then AD

$O(n)$

```
void grad_normalize(double[] out, double[] dout,
                    double[] in, double[] din) {
  double res = magnitude(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i] / res;
  }
  double d_res = 0;
  for (int i= N; i>=0; i--) {
    dres += -in[i]*in[i]/res * dout[i];
    din[i] += dout[i]/res;
  }
  grad_magnitude(in, din, n, dres);
}
```

Vector Normalization: AD, then LICM

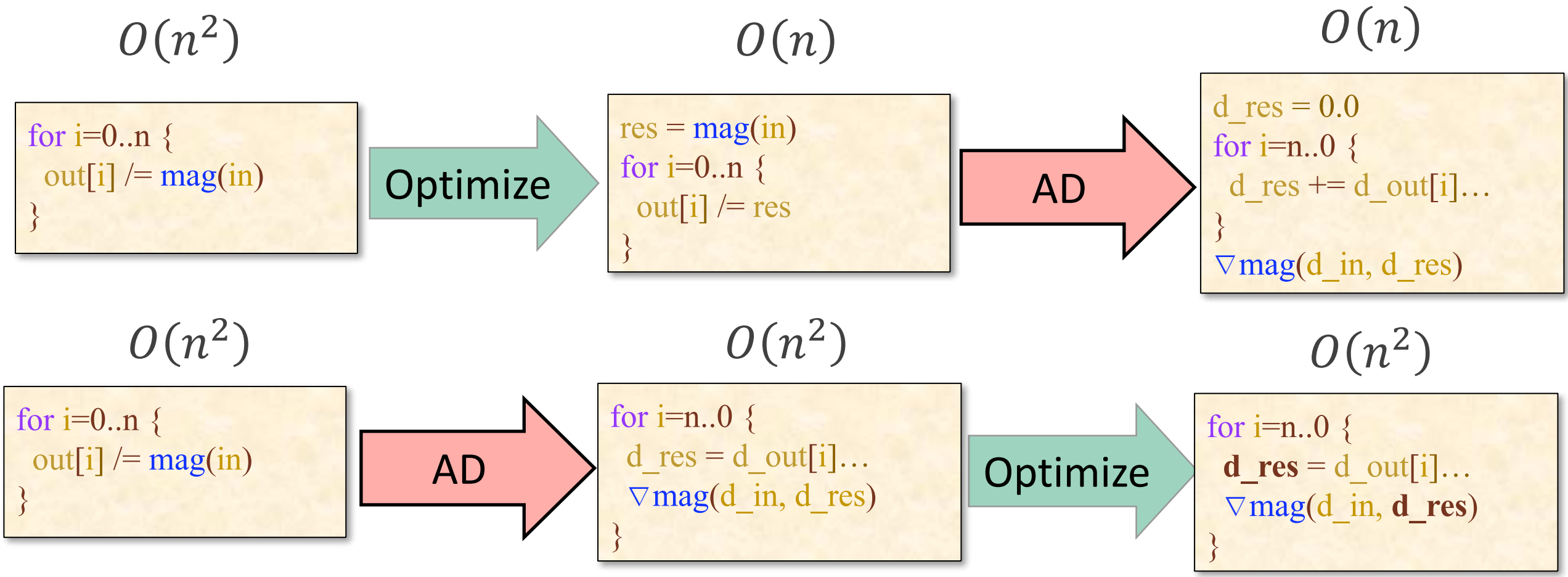
$O(n^2)$

```
void grad_normalize(double[] out, double[] dout,
                    double[] in, double[] din) {
  double res = magnitude(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i] / res;
  }
  for (int i= N; i>=0; i--) {
    double dres = -in[i]*in[i]/res * dout[i];
    din[i] += dout[i]/res;
    grad_magnitude(in, din, n, dres);
  }
}
```

Can't LICM as uses loop-local variable dres

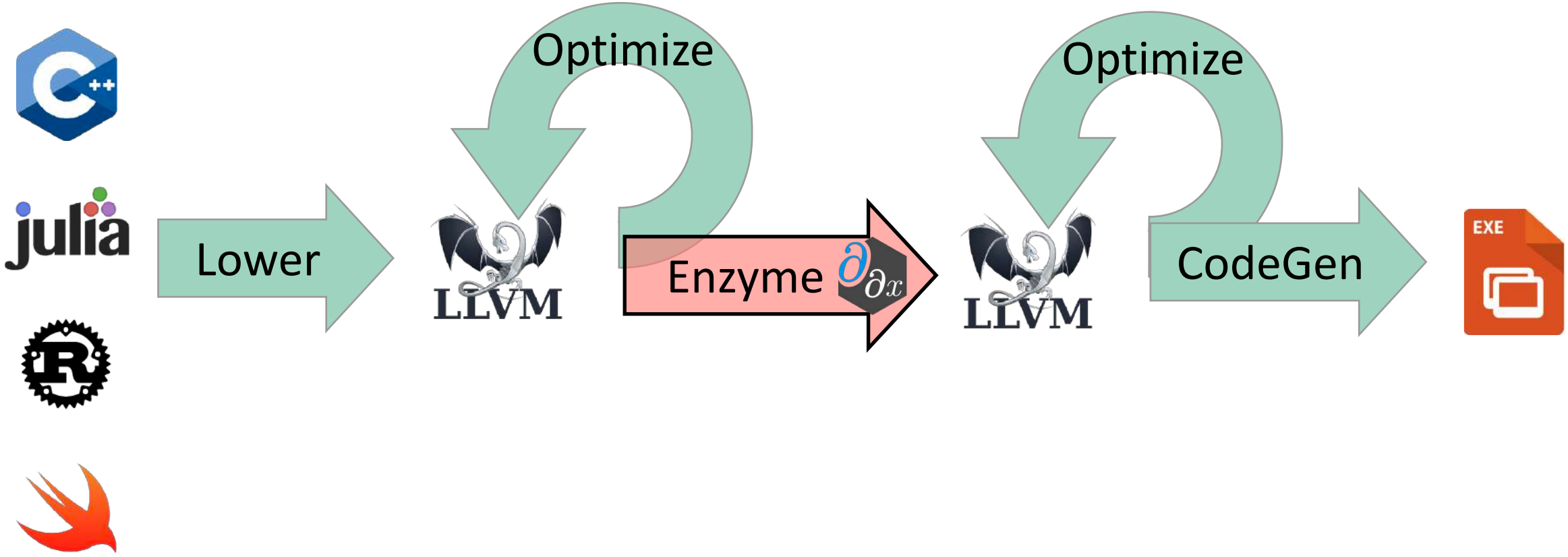
Optimization & Automatic Differentiation

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



Enzyme Approach [MC20]

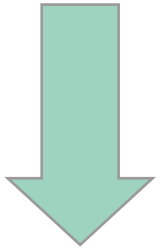
Performing AD at low-level lets us work on ***optimized*** code!



[MC20] Moses, Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. NeurIPS, 2020.

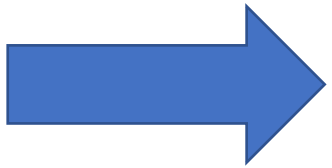
Case Study 2: Enzyme – AD of LLVM IR

```
double square(double val) {
  return val * val;
}

double __enzyme_autodiff(void*, ...);

double grad_square(double val) {
  return __enzyme_autodiff((void*)square, val);
}
```



Clang

```
define double @square(double %val) {
  %sq = fmul double %val, %val
  ret double %sq
}
```



```
define double @grad_square(double %val) {
  %res = fadd double %val, %val
  ret double %res
}
```

Case Study 2: Enzyme – AD of LLVM IR

```
double square(double val) {
  return val * val;
}

double __enzyme_autodiff(void*, ...);

double grad_square(double val) {
  return __enzyme_autodiff((void*)square, val);
}
```

Clang

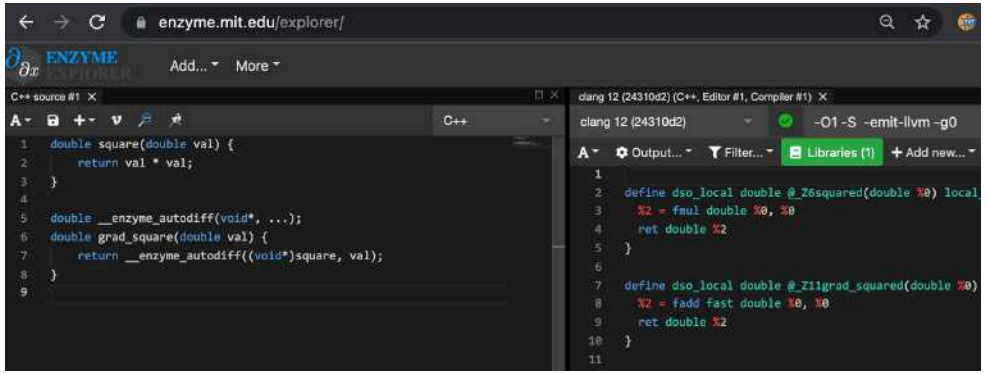
```
define double @square(double %val) {
  %sq = fmul double %val, %val
  ret double %sq
}
```

Enzyme

```
define double @grad_square(double %val) {
  %res = fadd double %val, %val
  ret double %res
}
```

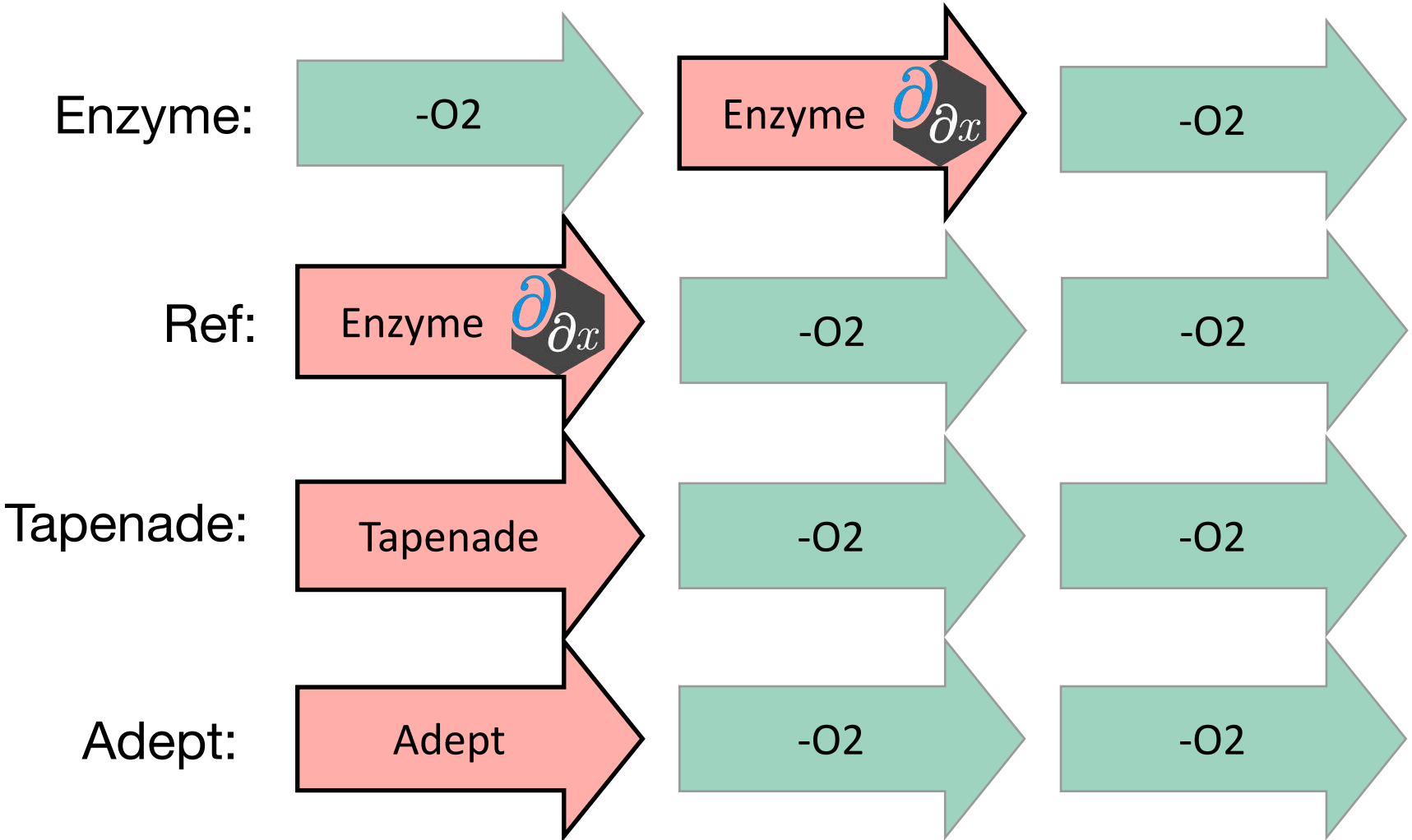
Try Online On The Enzyme Compiler Explorer!

https://bit.ly/3aNP6bB

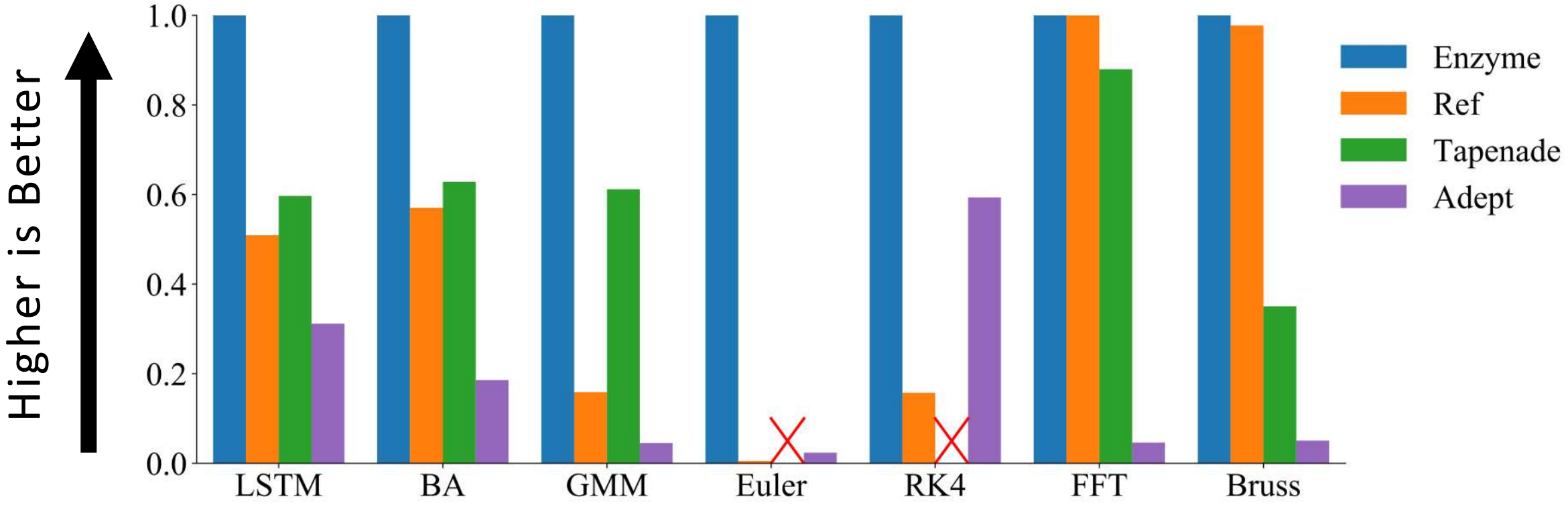


Enzyme Evaluation

Compared against Enzyme without preprocessing optimizations and two fastest AD tools



Speedup of Enzyme



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

Key  Enzyme Insights

- Running AD after/alongside optimization enables substantial speedups, including 4.2x on a suite of ML/scientific codes
- Enzyme achieves state-of-the art performance
- Enzyme is the first AD tool to differentiate arbitrary GPU kernels (including AMD and NVIDIA) [MCPH+21]
- Enzyme has support for generic forms of parallelism including OpenMP, MPI, and other frameworks that build upon them like Kokkos and RAJA

https://enzyme.mit.edu / https://github.com/wsmoses/Enzyme

[MCPH+21] Moses et al. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. *To appear at* SC, 2021.

Overall AD Compiler Insights

- Existing code does not need to be rewritten to be differentiated.
- Being within the compiler AD tools to continue function as the frontend languages & standards evolve.
- Has access to source locations and can issue precise diagnostics
- Can be successfully implemented at either a high or low level

…but this requires using a conformant compiler

=> Can we standardize this?

Standardization Efforts

- We believe first class support of differentiable programming paradigm is an important feature which will become central for various data science, research and industry communities
- We believe that compiler-aided AD is the most viable path forward to supporting high-performance
- We have produced an overview paper “Differentiable programming for C++” https://wg21.link/P2072
- We have solicited feedback from the ML study group of isocpp (aka SG19) but also from various other parties
- We are keen on turning the overview paper into a concrete plan!

Conclusion

- Differentiable Programming is a new and promising programming paradigm which relies on well developed theory and technology
- The presented tools are being developed and integrated in various fields
- The standardization efforts are ramping up and we hope to solicit support after this talk

Thank you!

Q & A