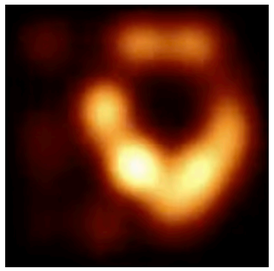
William (Billy) Moses; Asst Prof @ UIUC

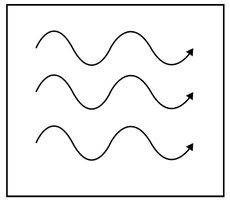
- Lead the PRONTO Lab
- How do we enable domain-experts to focus on their intended problem instead having to also become expert in performance, parallelism, ML, mathematics, security, & more?

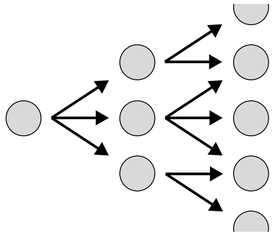


>100x speedup!

Prior:
5 days (cluster)

Enzyme-Based:
1 hour (laptop)





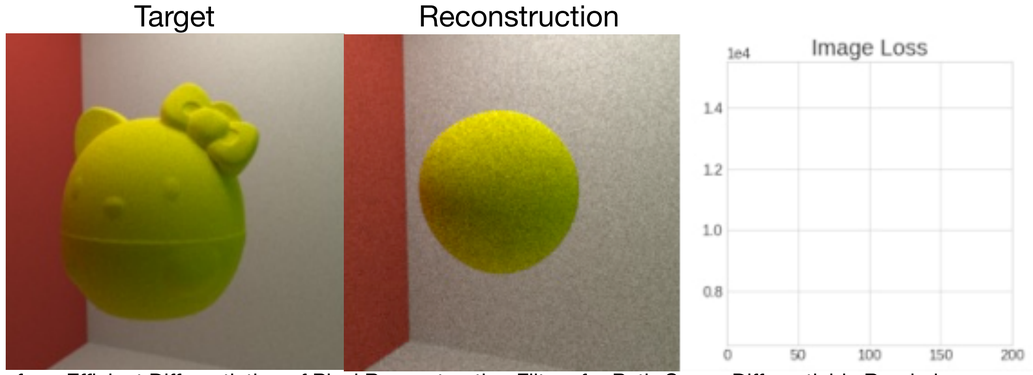
Efficient Differentiation of Arbitrary Programs [1] [2] [3]

Synthesize GPU & parallel programs with Polygeist/MLIR [4] [5] [6]

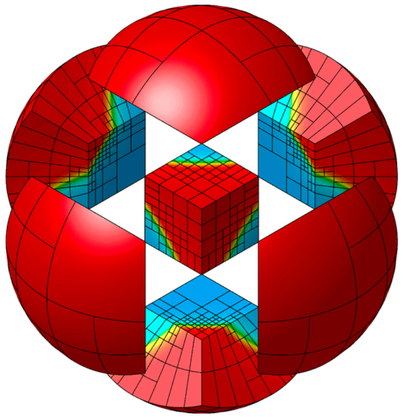
Use ML to discover the fastest programs [7] [8] [9]

automatically secure programs (FHE) [10], database compilation, & lots more fun!

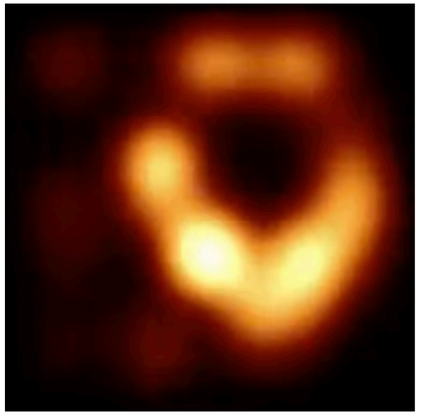
AD-Powered Applications



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

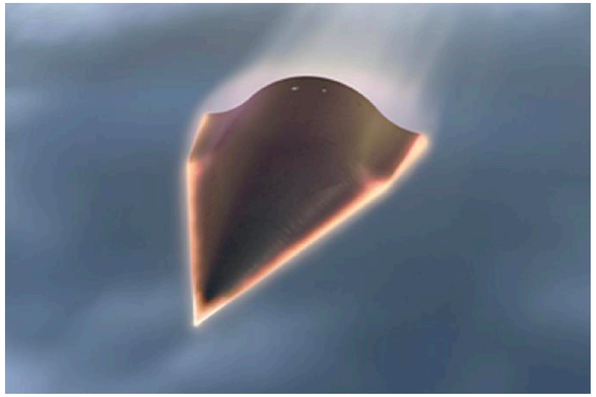


from MFEM Team at LLNL

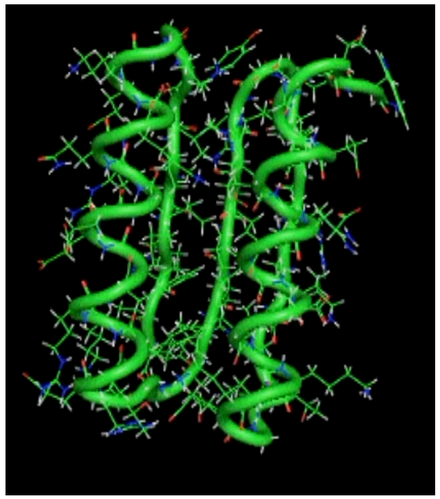


from Comrade: High Performance Black-Hole Imaging JuliaCon 2022, Paul Tiede (Harvard)

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



from Center for the Exascale Simulation of Materials in Extreme Environments

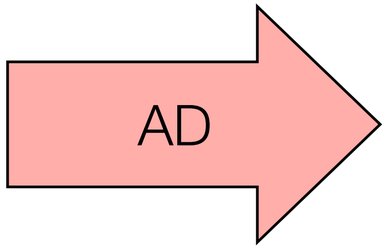


from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs

```
function relu3(x::Float64)
  if x > 0
    return x^3
  else
    return 0
  end
```




```
function grad_relu3(x::Float64)
  if x > 0
    return 3*x^2
  else
    return 0
  end
```

- Unlike numerical approaches, automatic differentiation (AD) can compute the derivative of ALL inputs (or outputs) at once, without approximation error!

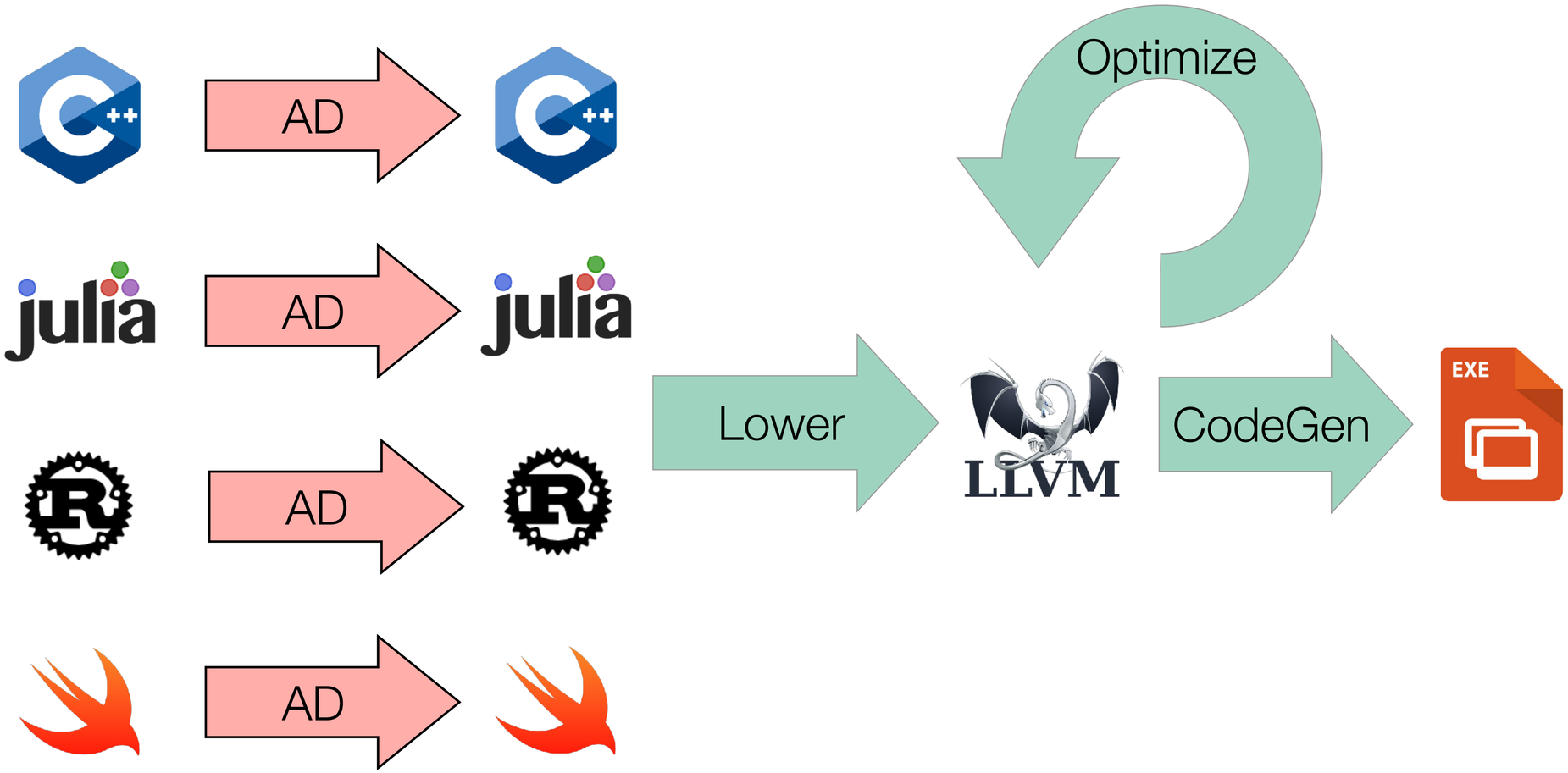
```
// Numeric differentiation
// f'(x) approx [f(x+epsilon) - f(x)] / epsilon
grad_input = []

for i in 1:100
  input2 = copy(input)
  input2[i] += 0.01;
  push!(grad_input, (f(input2) - f(input))/0.001)
end
```

```
// Automatic differentiation
grad_input = zeros(input)

Enzyme.autodiff(Reverse, f,
   Duplicated(input, grad_input))
```

Existing Automatic Differentiation Pipelines



Optimization & Automatic Differentiation

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

```
for i = 1:n
 out[i] /= mag(in)
end
```

Optimize

$O\left(n\right)$

```
res = mag(in)
for i = 1:n
 out[i] /= res
end
```

AD

$O\left(n\right)$

```
d_res = 0.0
for i = n:1
  d_res += d_out[i]…
end
∇mag(d_in, d_res)
```

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

```
for i = 1:n
 out[i] /= mag(in)
end
```

AD

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

```
for i = n:1
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
end
```

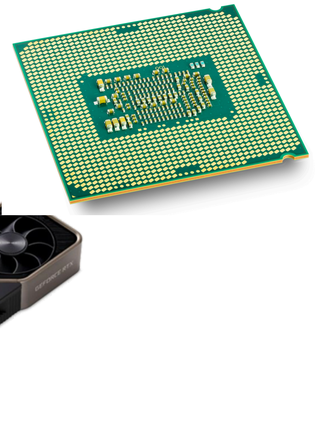
Optimize

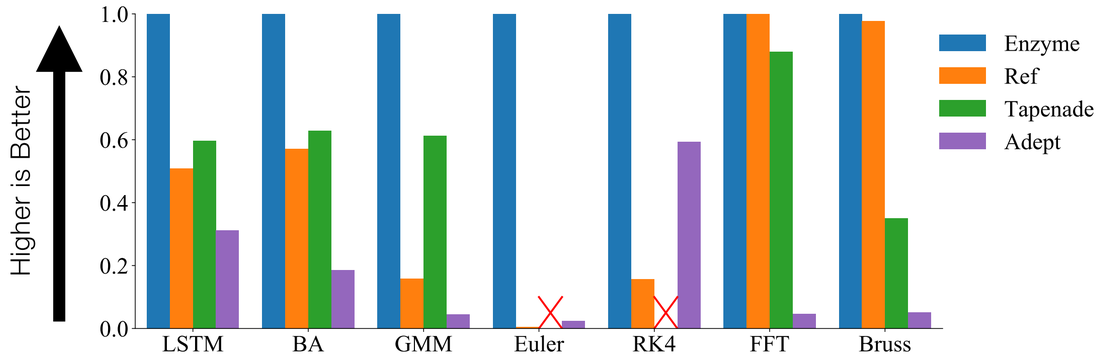
$O\left(n^2\right)$

```
for i = n:1
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
end
```

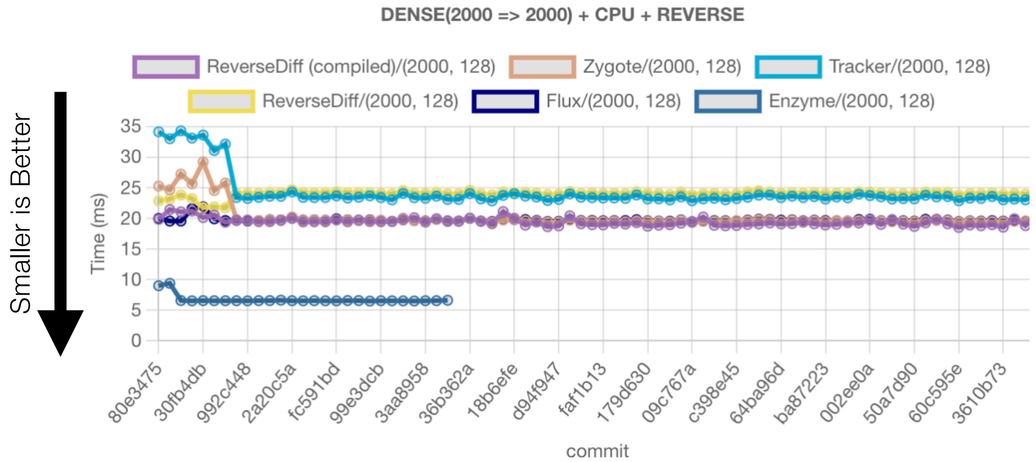
Enzyme.jl:

- Automatic Differentiation in the compiler and run alongside optimization
- Forward and Reverse Mode AD, including batched versions of both
- Handles mutation, parallelism, GPU's (AMD, CUDA, etc), & more!
- Static analysis & optimization => very, *very* fast scalar & vector AD
- Joint with Valentin Churavy and many others





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



Enzyme.jl

```
function taylor(x, N)
  sum = 0 * x
  for i = 1:N
    sum += x^i / i
  end
  return sum
end
```

```
def taylor_jax(x, N):
  sum = 0 * x
  for i in range(1,N):
    sum += x**i / i
  return sum
```

```
def taylor_lax(x, N):
  return jax.lax.fori_loop(
       1,
       N,
       lambda i, cur:
          cur + x**i / I,
       0)
```

```
@btime Enzyme.autodiff(Forward, taylor, Duplicated(0.5, 1.0), 10^6)
#      30 ms (0 bytes)

@btime Enzyme.autodiff(Reverse, taylor, Active(0.5), 10^6).
#      30 ms (0 bytes)

@btime ForwardDiff.derivative(x -> taylor(x, 10^6), 0.5)
#      60 ms (0 bytes)

@btime Zygote.gradient(taylor, 0.5, 10^6)
#     993 ms (663.56 MiB)

@btime gradient(x -> taylor(x, 10^6), MoonCake(safe_mode=false), 0.5)
#    4149 ms (117.05 MiB). [preparation]
#     391 ms (16 byte).    [evaluation]

@btime Diffractor.gradient(taylor, 0.5, 10^6)
#   96665 ms (96.37 GiB)

@pytime jax.grad(taylor_jax)(0.5, 10^5)
# >183993 ms

@pytime jax.grad(taylor_lax)(0.5, 10^6)
#      95 ms
```

Reactant.jl (we really need a logo)

- Compile your Julia code to remove type-instabilities, unnecessary allocations, fuse kernels, and beyond!
- Execute your code on CPU, GPU, and TPU without rewriting thanks to OpenXLA!
- Of course compatible with Enzyme, Lux, etc
- joint with Sergio Sanchez, Avik Pal, Paul Berg, Jules Merckx, Julian Samaroo + others

```julia
input1 = Reactant.to_rarray(ones(10))
input2 = Reactant.to_rarray(ones(10))

function sum_sinadd(x, y)
   return sum(sin.(x) .+ y)
end

f = @compile sum_sinadd(input1, input2)

# A much faster version of the code, and
#  automatically on your favorite device
f(input1, input2)
```

CUDA KAN network

Forward (regular Julia)	Backwards (Zygote + Julia)
47.586 us (248 allocations)	289.319 us (575 allocations)
234.233 us (1022 allocations)	2099.000 us (1055 allocations)
134.028 us (668 allocations)	1772.000 us (877 allocations)

Forward (Reactant)	Backwards (Enzyme + Reactant)
39.873 us (2 allocations)	51.691 us (3 allocations)
68.439 us (6 allocations)	104.193 us (3 allocations)
55.889 us (6 allocations)	80.020 us (3 allocations)