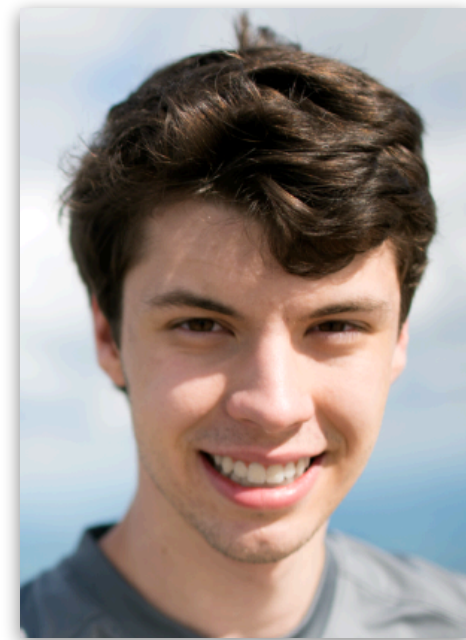


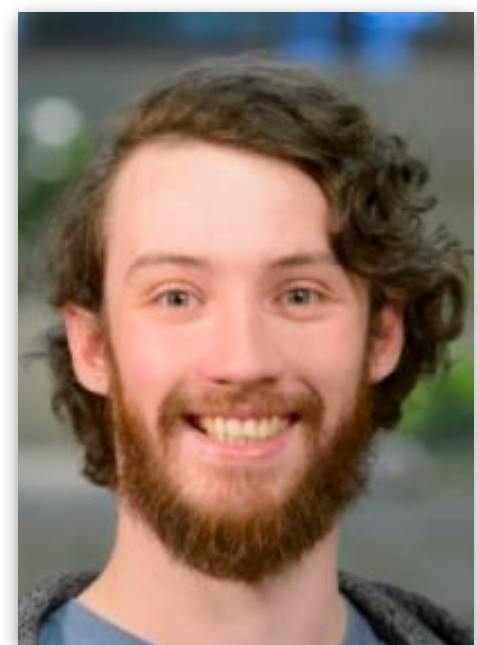
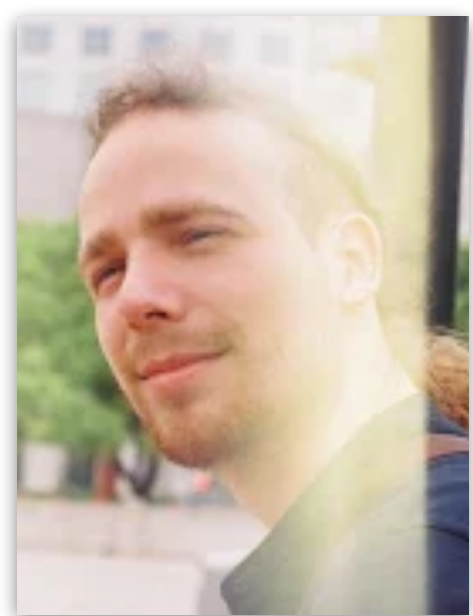
# Supercharging Programming Through Compiler Technology



William S. Moses

wmoses@mit.edu  
MIT Thesis Defense  
May 1, 2023





Valentin Churavy

Leila Ghaffari

Ludger Paehler

Johannes  
Doerfert

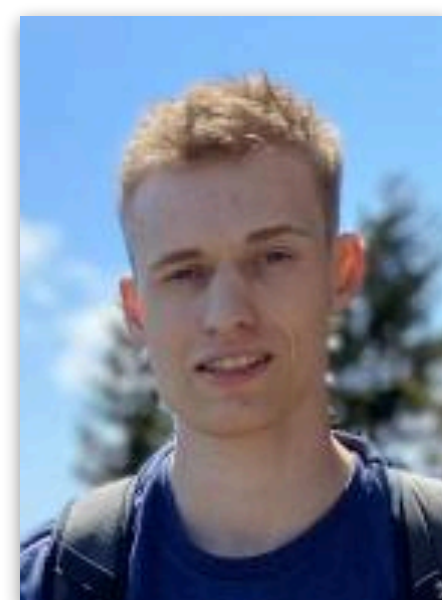
Jan Hückelheim

Charles E.  
Leiserson

Zach Devito

Andrew Adams

Lorenzo  
Chelini



Sri Hari Krishna  
Narayanan

Michel  
Schanen

Paul Hovland

TB Schardl

Praytush Das

Tim Gymnich

Albert Cohen

Sven  
Verdoolaege

Ruizhe Zhao



Manuel  
Drehwald

Nicolas  
Vasliache

Alex Zinenko

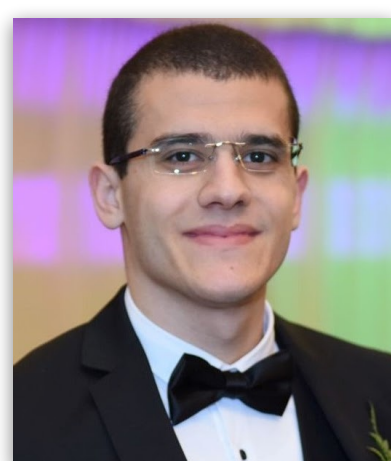
Theodoros  
Theodoridis

Priya Goyal

Ivan R. Ivanov

Jens Domke

Toshio Endo



Ameer  
Haj Ali

Jenny  
Huang

Ion  
Stoica

Krste  
Asanovic

John  
Wawrzynek

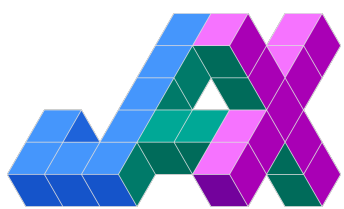
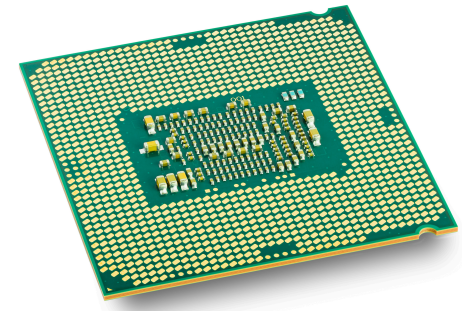
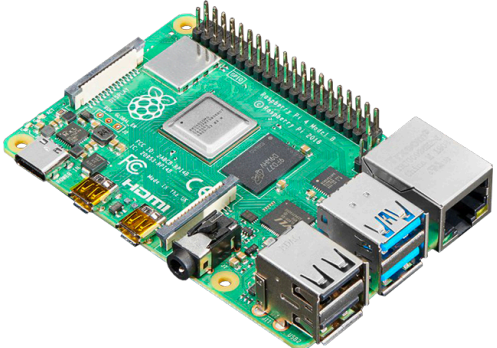
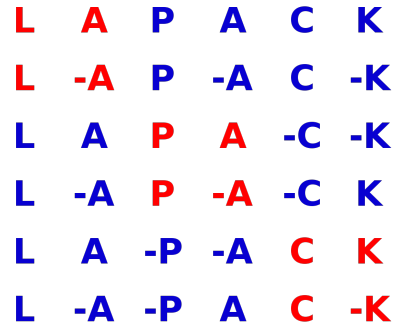
&

more



# The Programmer's Burden

- The decline of Moore's law and an increasing reliance on computation => explosion of specialized software packages and hardware architectures.
- Domain-experts must customize programs and learn platform-specific API's, instead of working on their intended problem.
- Rather than each user bearing this burden, compilers can automatically generate fast, portable, and composable programs!

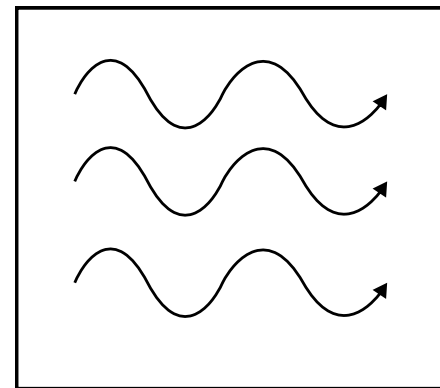




# Extending the Boundaries of Compilers



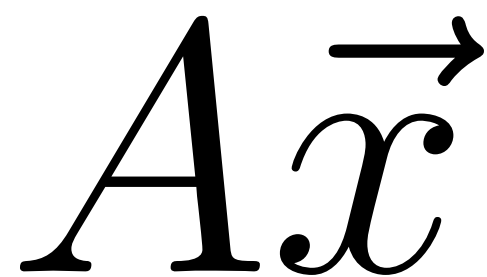
Enzyme: fast, parallel, and rewrite-free **derivative generation**; best student paper @SC'22, SC'21, spotlight @NeurIPS'20; awarded multi-year DOE grant with LLNL



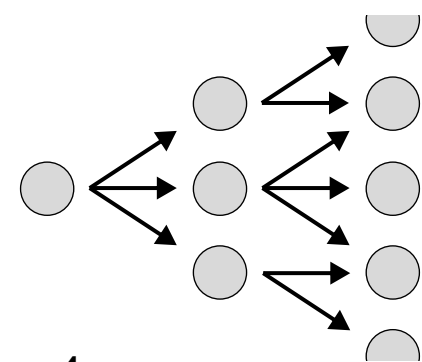
Tapir: understand and optimize **parallel programs**; *best paper @PPOPP'17, TOPC'19*



Polygeist: **run GPU code on CPUs**, 2.7x faster than expert-written code, preserve program structure to leverage device parameters perform HLS; PPOPP'23, PACT'21



Tensor Comprehensions (TC): automatically **generate fast tensor arithmetic**; TACO'19



AutoPhase: **ML-based optimization** of programs/circuits; MLSys'20, FCCM'19

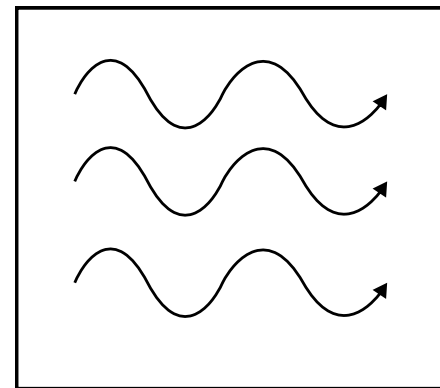


# Extending the Boundaries of Compilers

---



Enzyme: fast, parallel, and rewrite-free **derivative generation**; best student paper @SC'22, SC'21, spotlight @NeurIPS'20; awarded multi-year DOE grant with LLNL



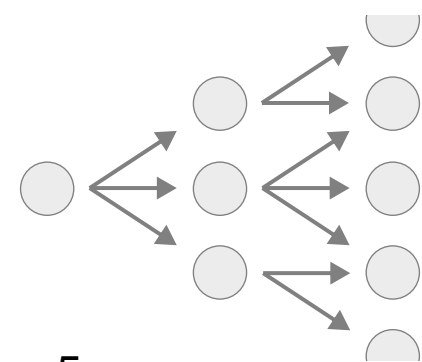
Tapir: understand and optimize **parallel programs**; *best paper @PPOPP'17, TOPC'19*



Polygeist: **run GPU code on CPUs**, 2.7x faster than expert-written code, preserve program structure to leverage device parameters perform HLS; PPOPP'23, PACT'21



Tensor Comprehensions (TC): automatically **generate fast tensor arithmetic**; TACO'19



AutoPhase: **ML-based optimization** of programs/circuits; MLSys'20, FCCM'19

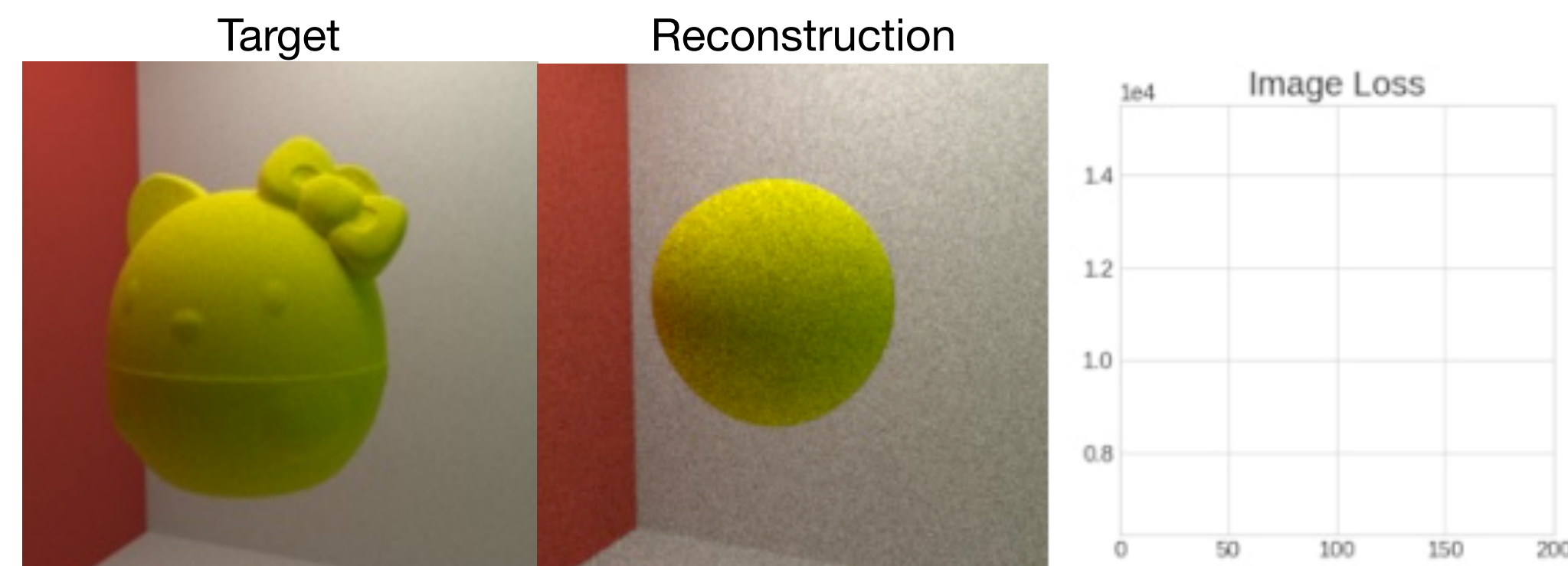
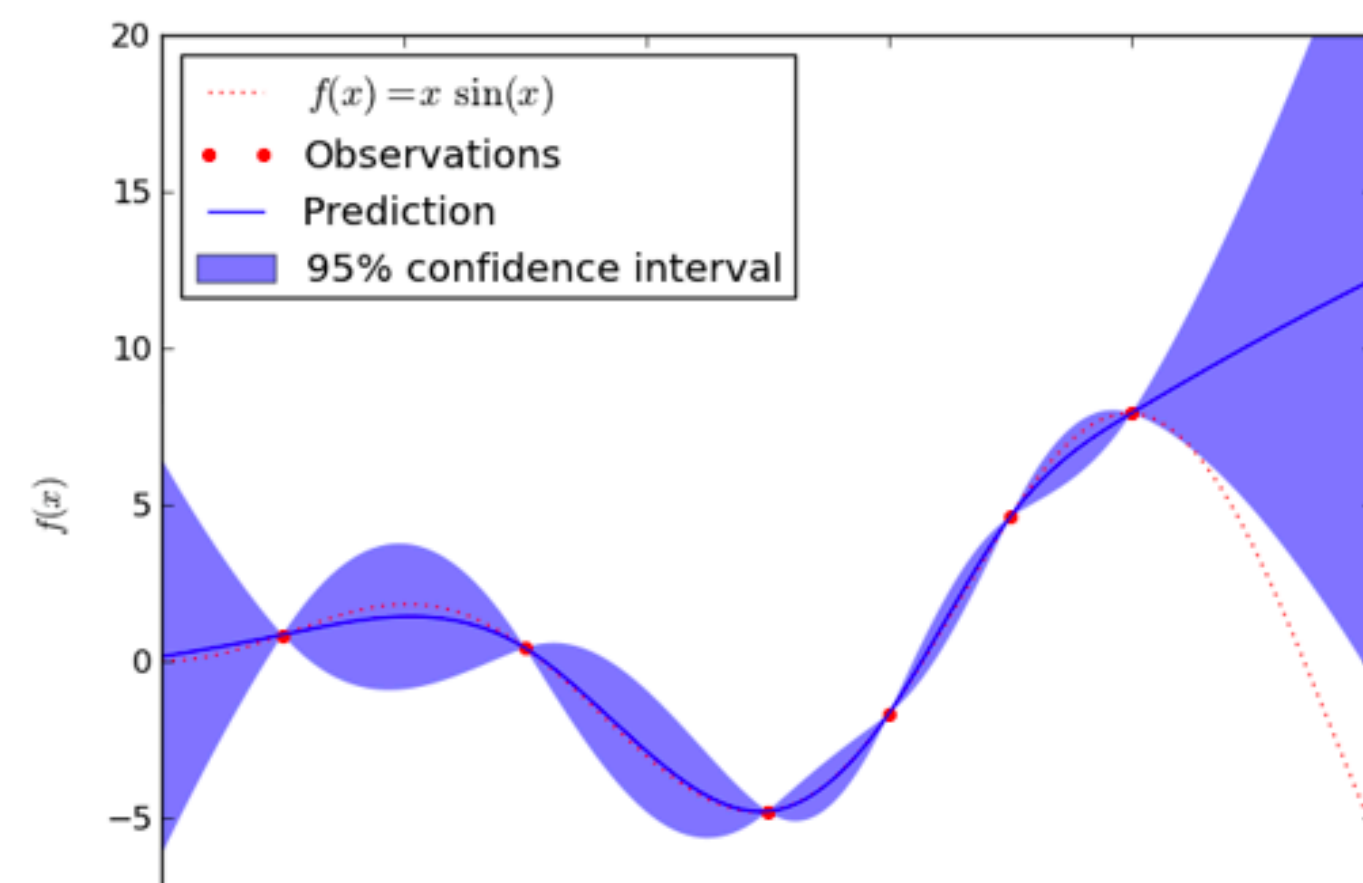


# AP Calculus: Revisited

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

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

- 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

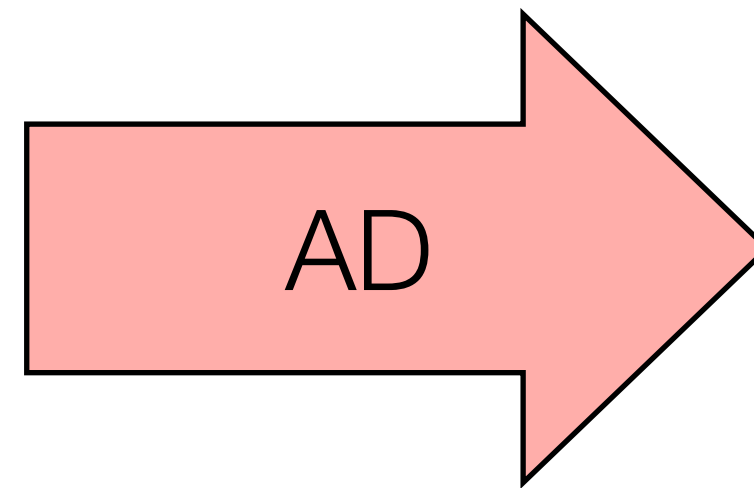




# Automatic Derivative Generation

- 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)
```



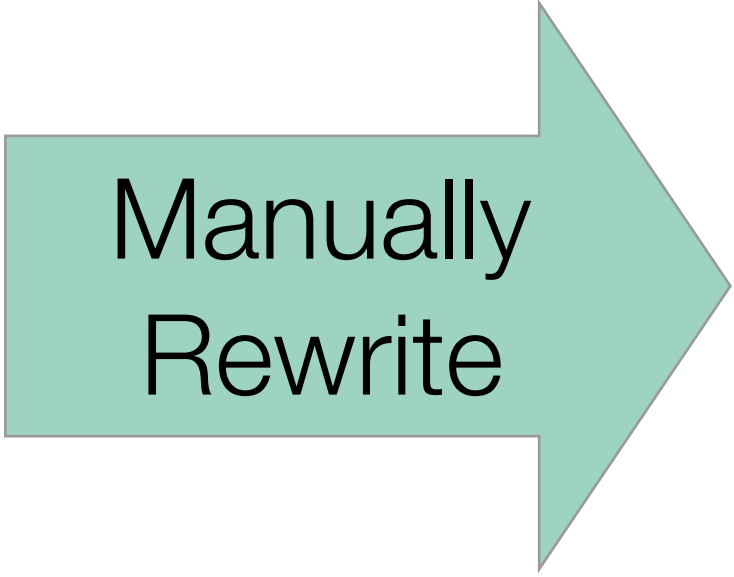
# Existing AD Approaches (1/3)

---

- 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;

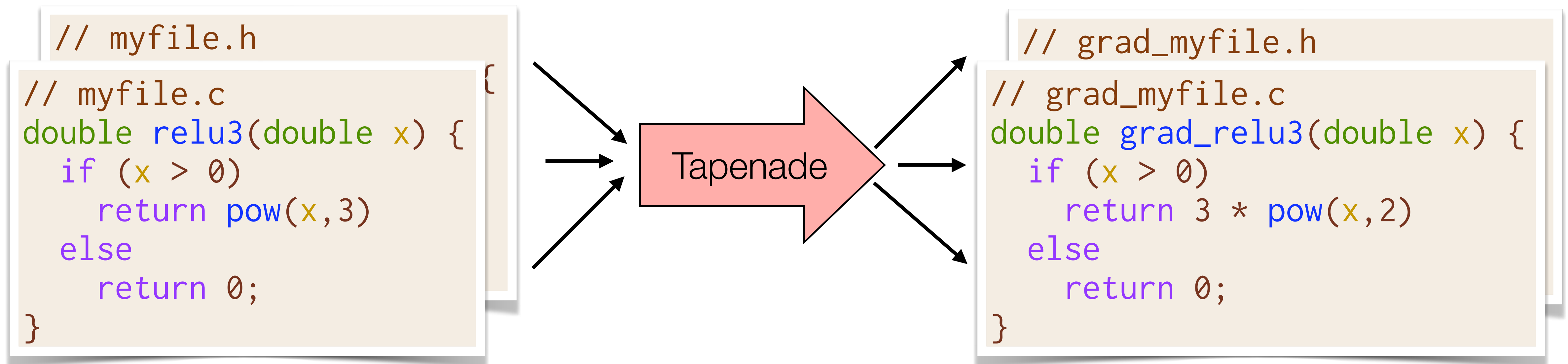
// Store all instructions into stack
adept::adouble out(relu3(inp));
out.set_gradient(1.00);

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



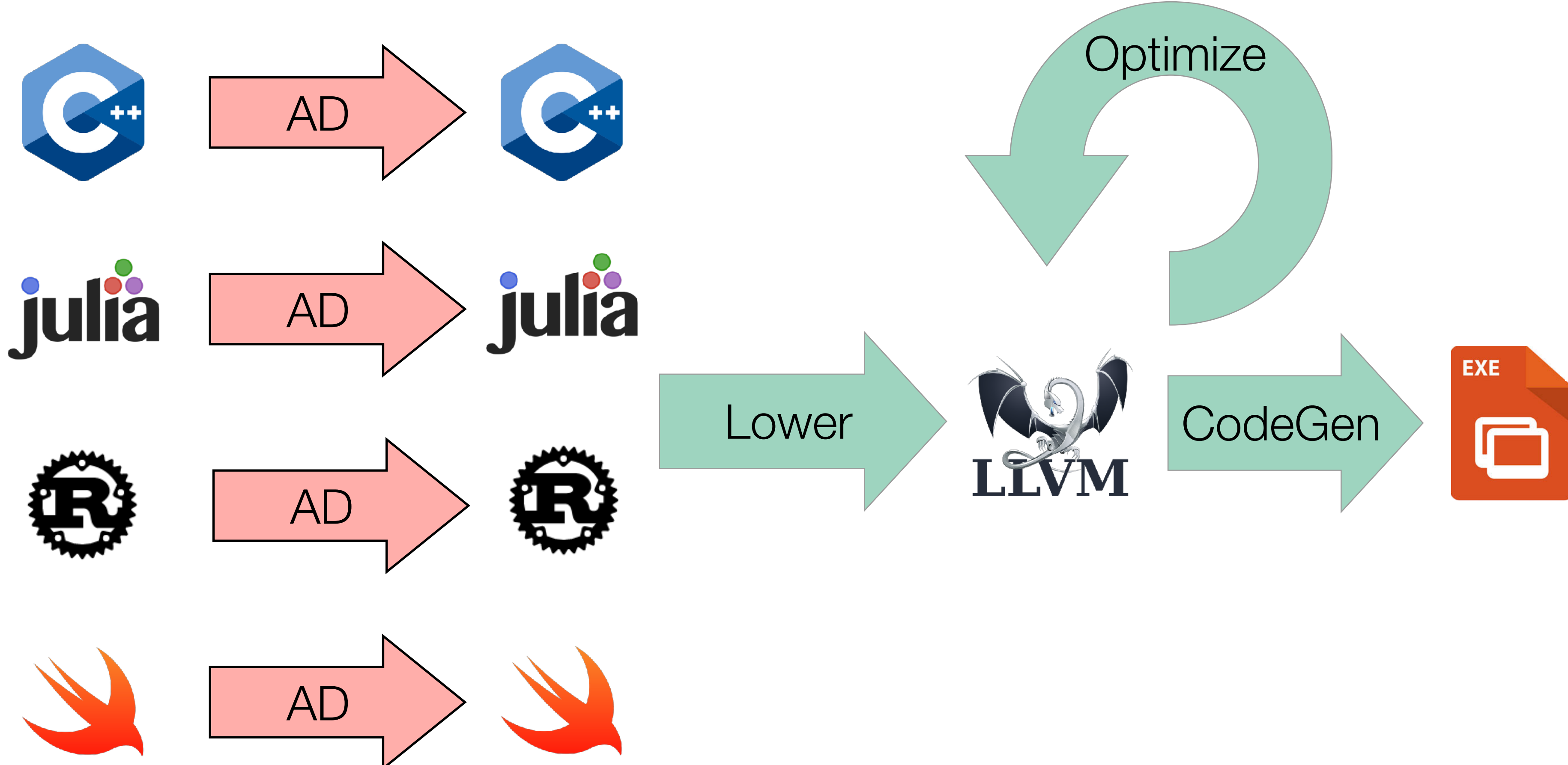
# Existing AD Approaches (3/3)

- 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



# 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);
    }
}
```

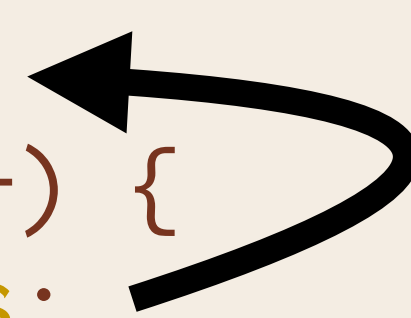


# Case Study: Vector Normalization

---

```
//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]...  
}  
∇mag(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

$O(n^2)$

```
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

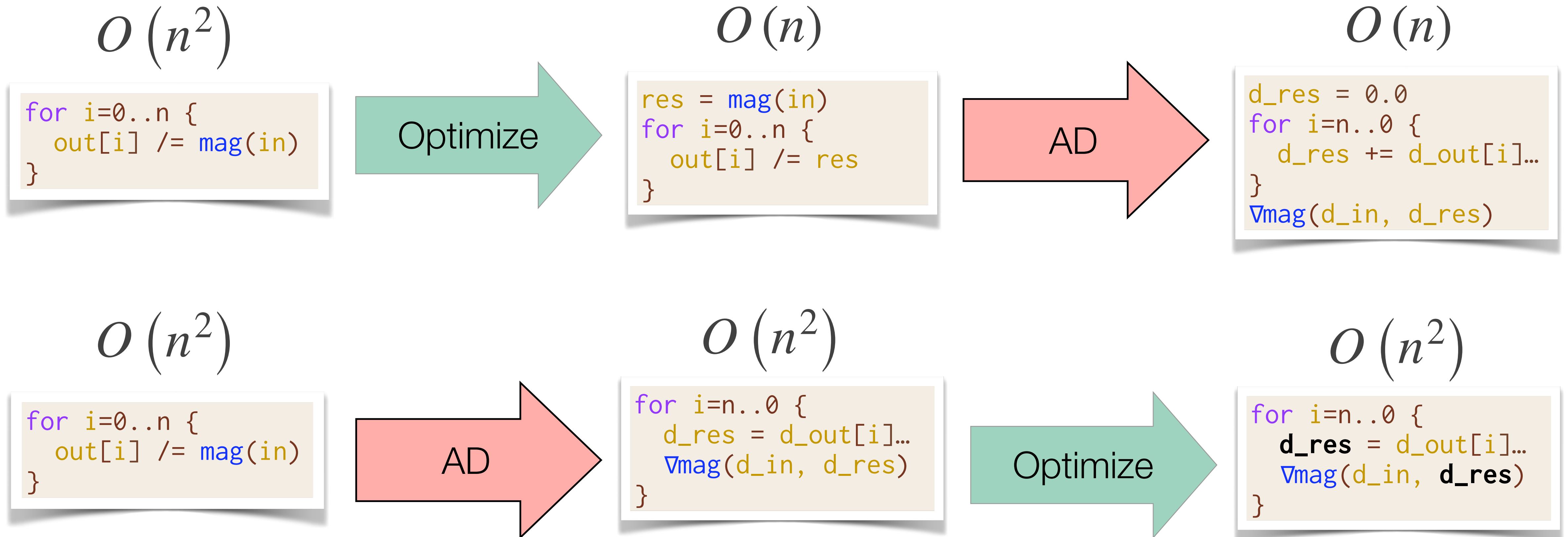
$O(n^2)$

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



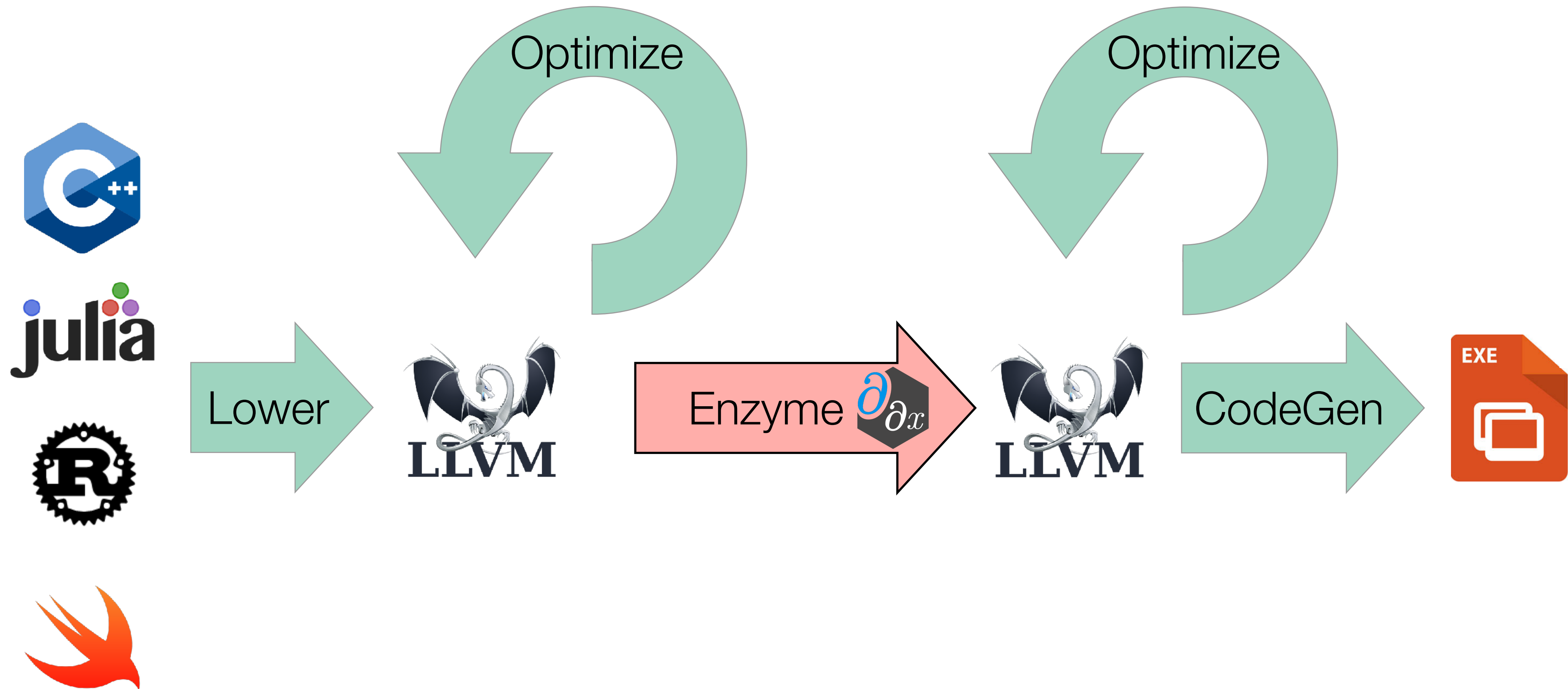
# Optimization & Automatic Differentiation

Differentiating after optimization can create *asymptotically faster* gradients!



# Enzyme Approach

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





# Case Study: ReLU3

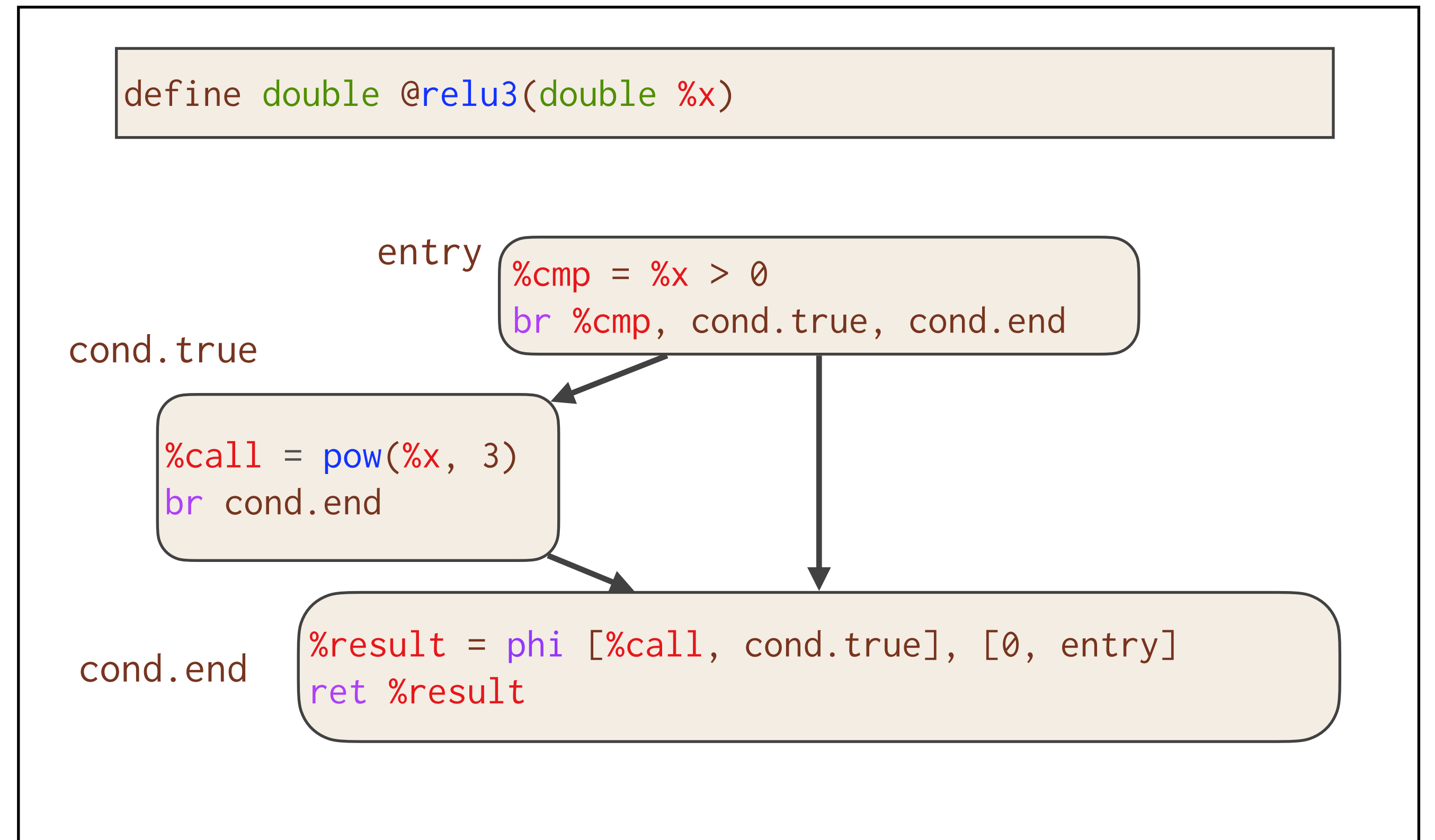
## C Source

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

## Enzyme Usage

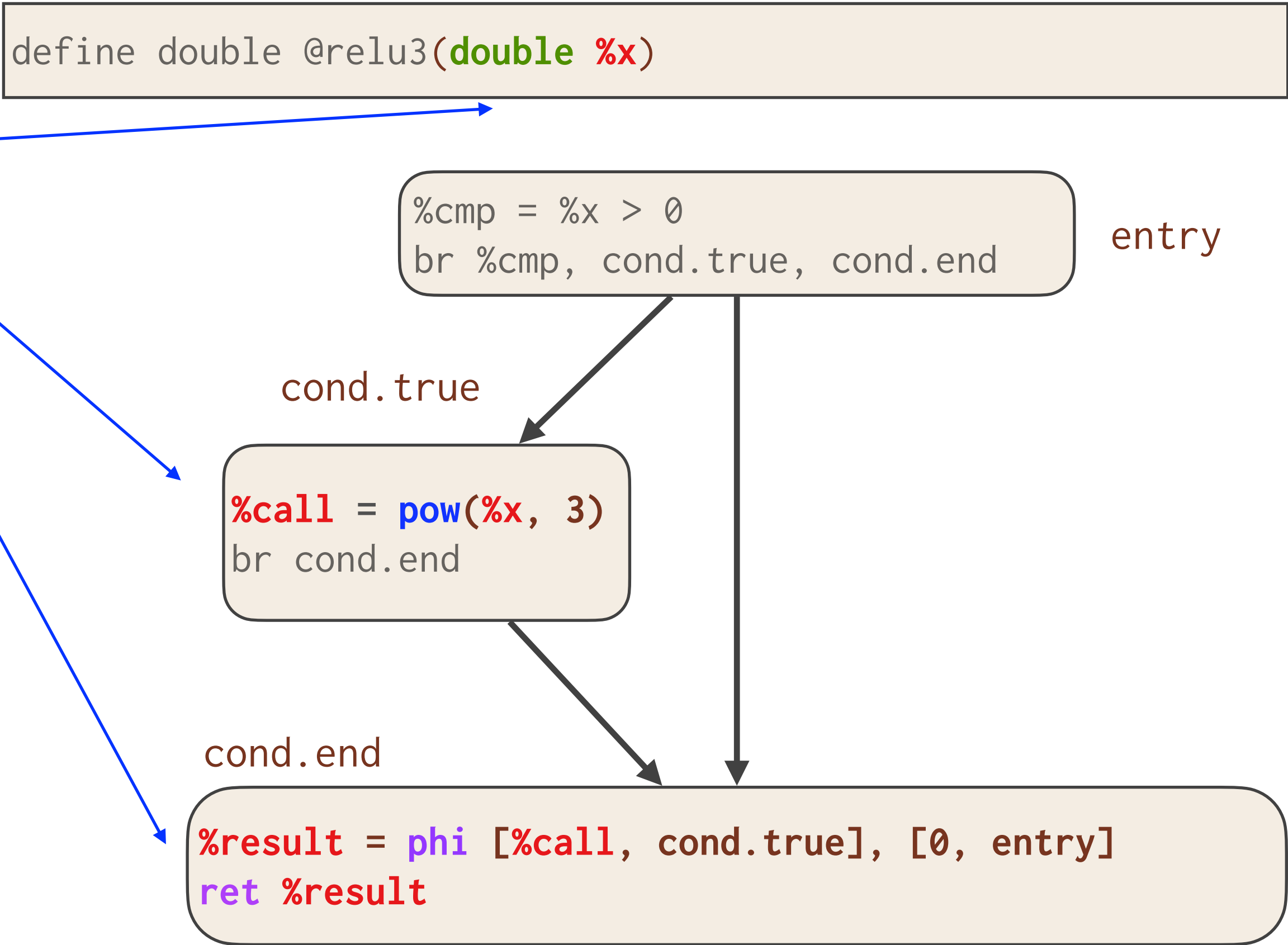
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

## LLVM

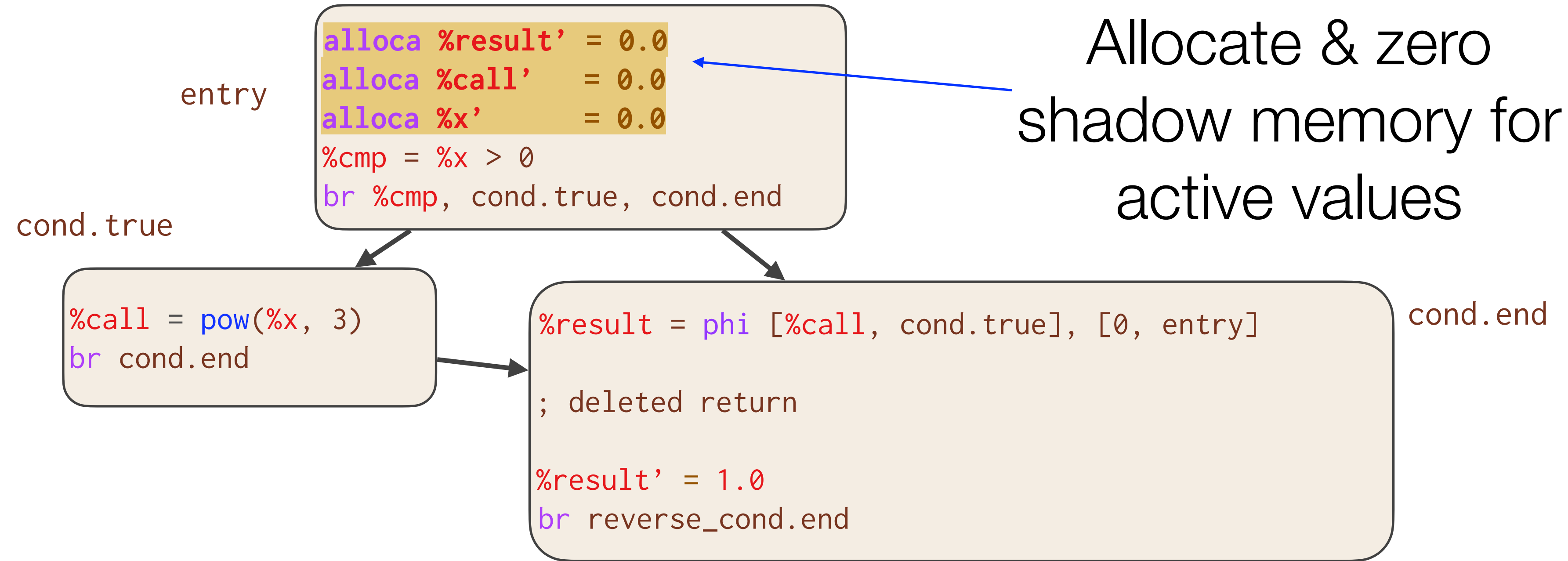


# Case Study: ReLU3

Active Instructions



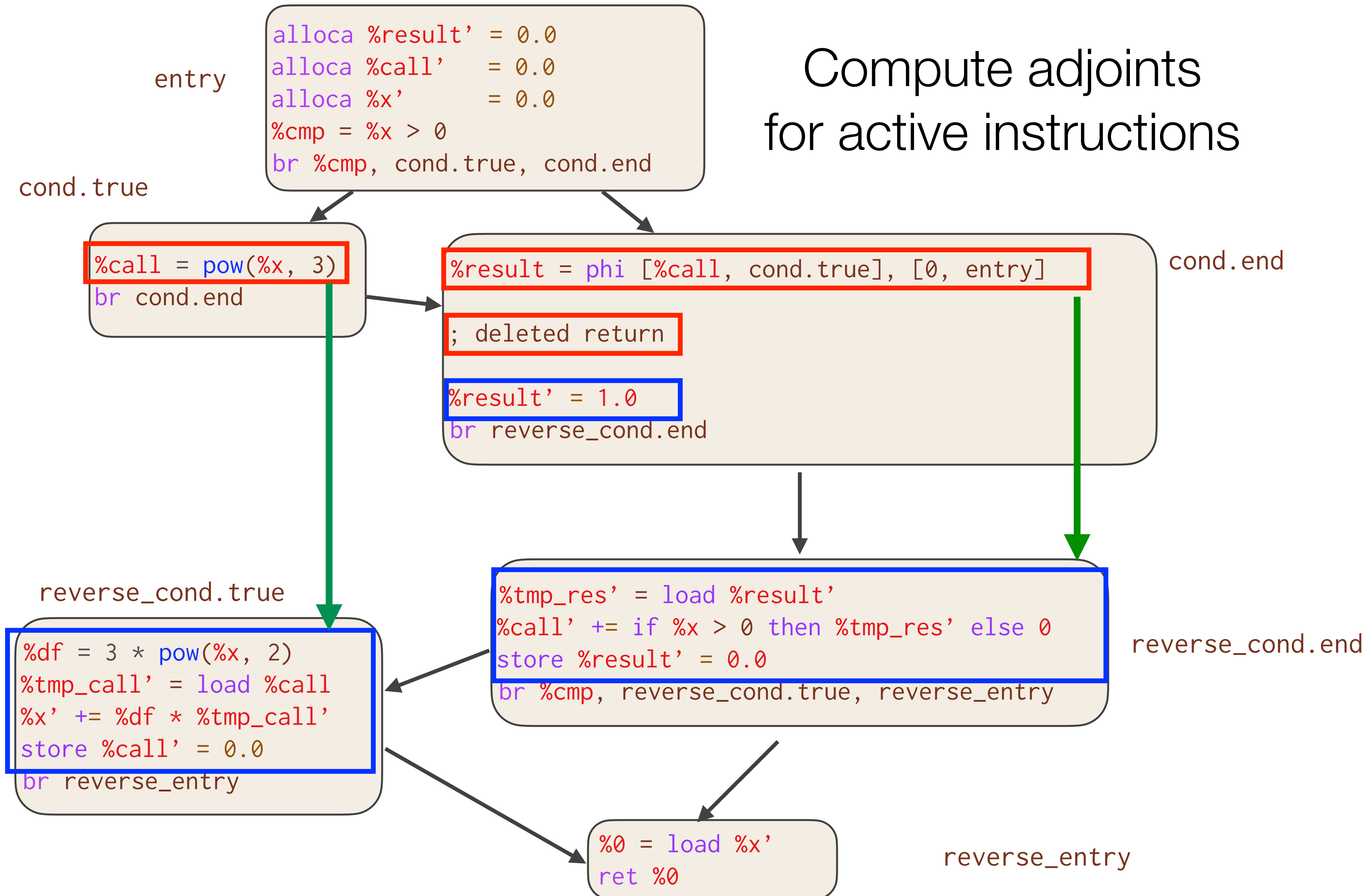
```
define double @diffe_relu3(double %x, double %differet)
```





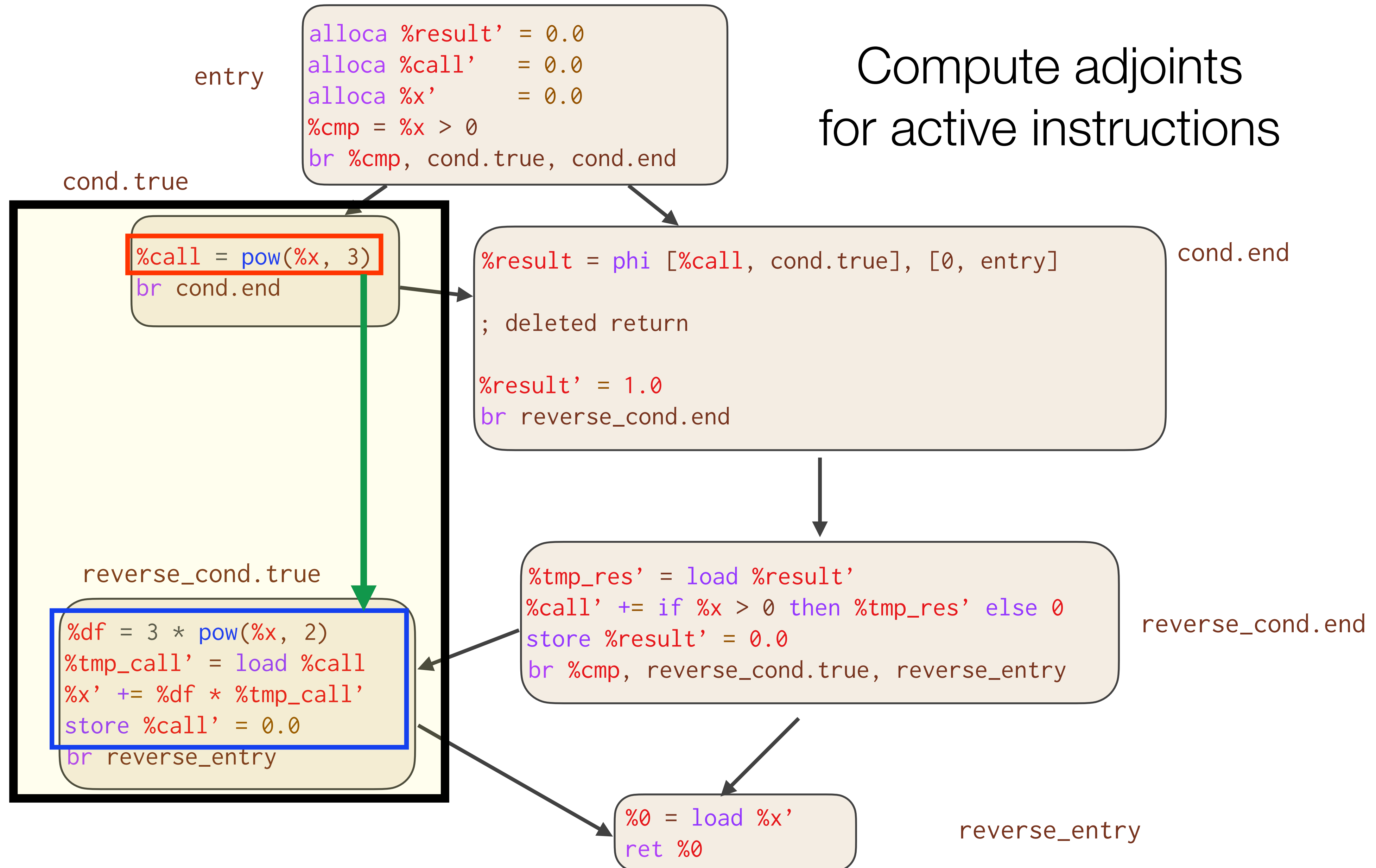
```
define double @diffe_relu3(double %x, double %differet)
```

## Compute adjoints for active instructions



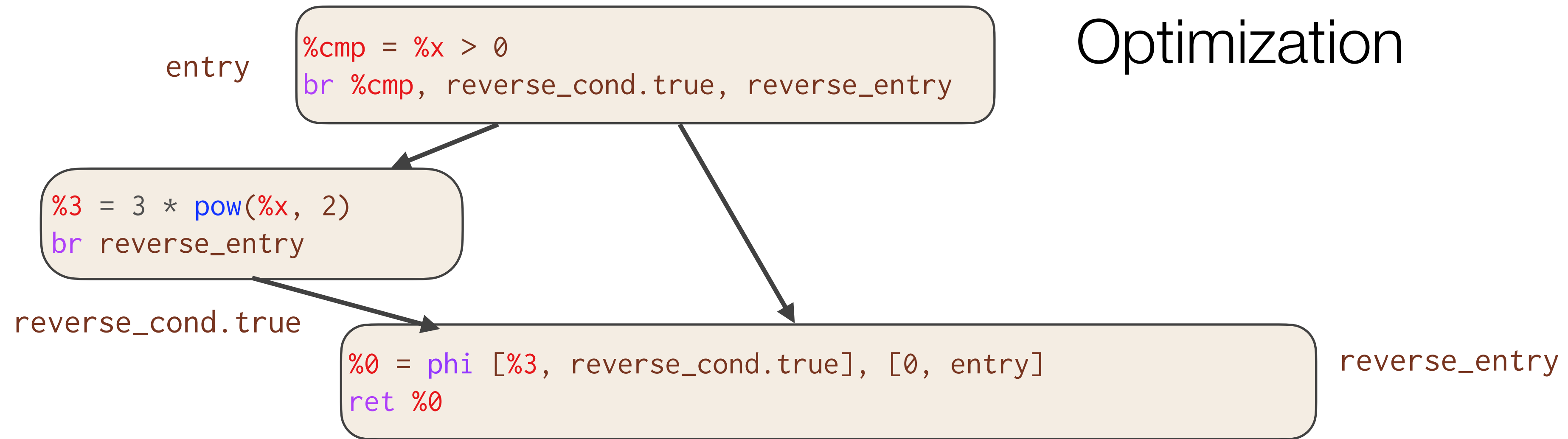
```
define double @diffe_relu3(double %x, double %differet)
```

# Compute adjoints for active instructions



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

## Post Optimization



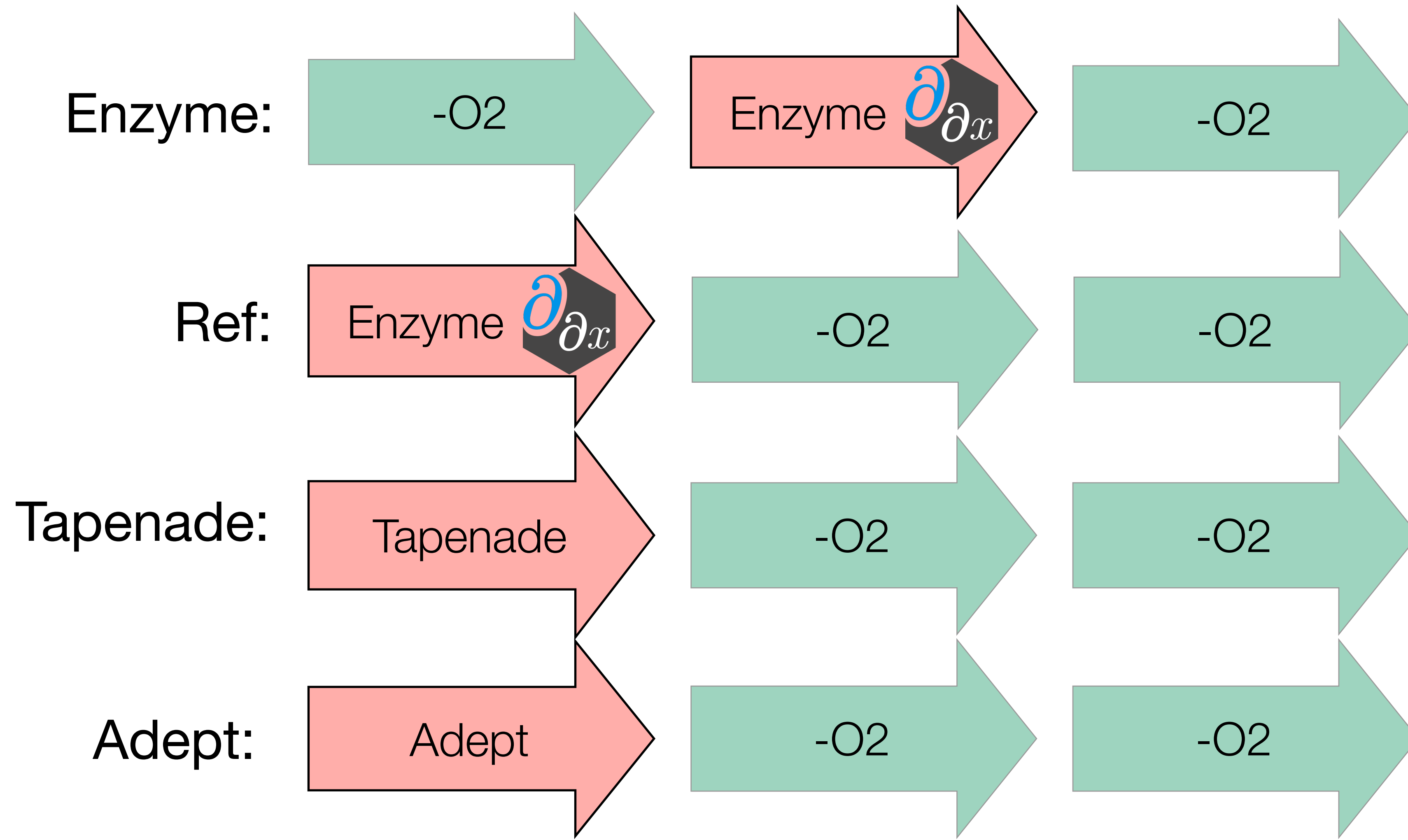
Essentially the optimal hand-written gradient!

```
double diffe_relu3(double x) {  
    double result;  
    if (x > 0)  
        result = 3 * pow(x, 2);  
    else  
        result = 0;  
    return result;  
}
```

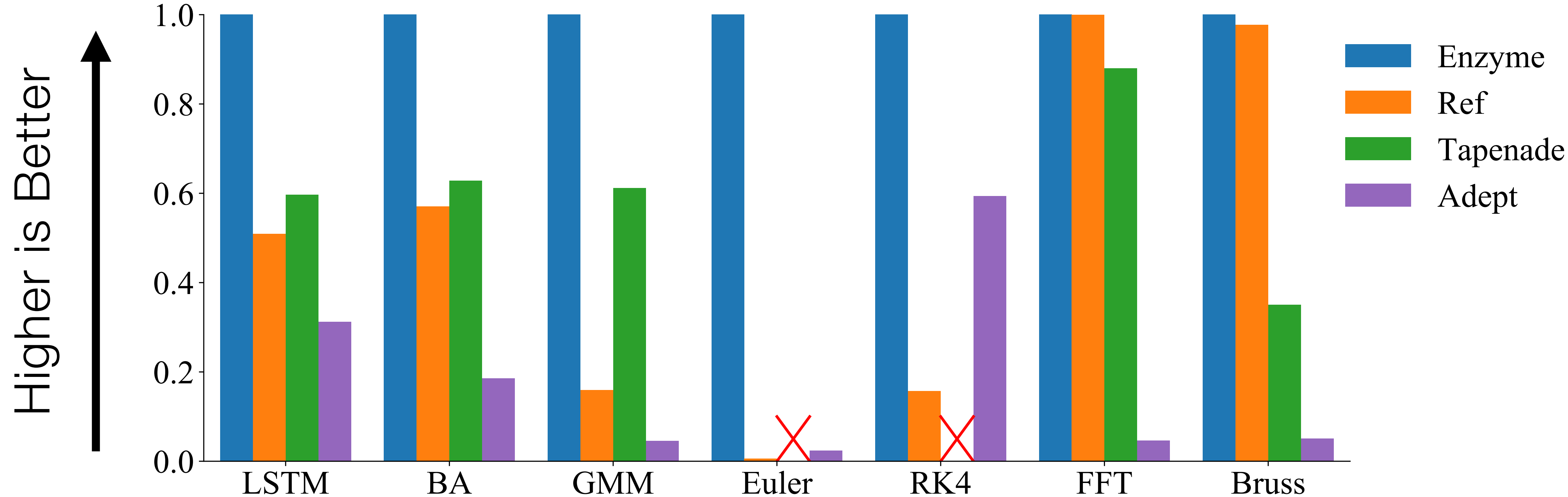


# Experimental Setup

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



# Speedup of Enzyme



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

# Automatic Differentiation & GPUs

---

- 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





# Efficient GPU Code

---

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

```
// Forward Pass
out[i] = x[i] * x[i];
x[i] = 0.0f;

// Reverse (gradient) Pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

# Efficient Correct GPU Code

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

```
double* x_cache = new double[...];

// Forward Pass

out[i] = x[i] * x[i];
x_cache[i] = x[i];

x[i] = 0.0f;

// Reverse (gradient) Pass

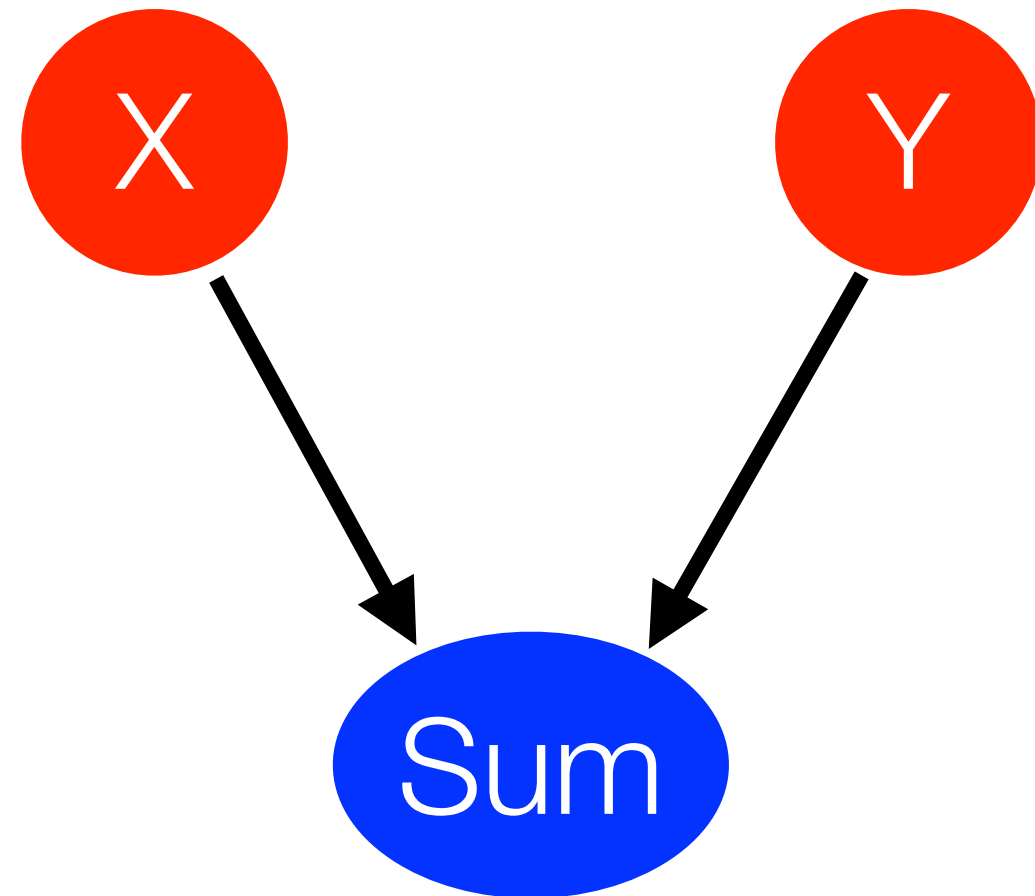
...
grad_x[i] += 2 * x_cache[i]
             * grad_out[i];
...

delete[] x_cache;
```

# Cache Reduction Example

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

Overwritten:



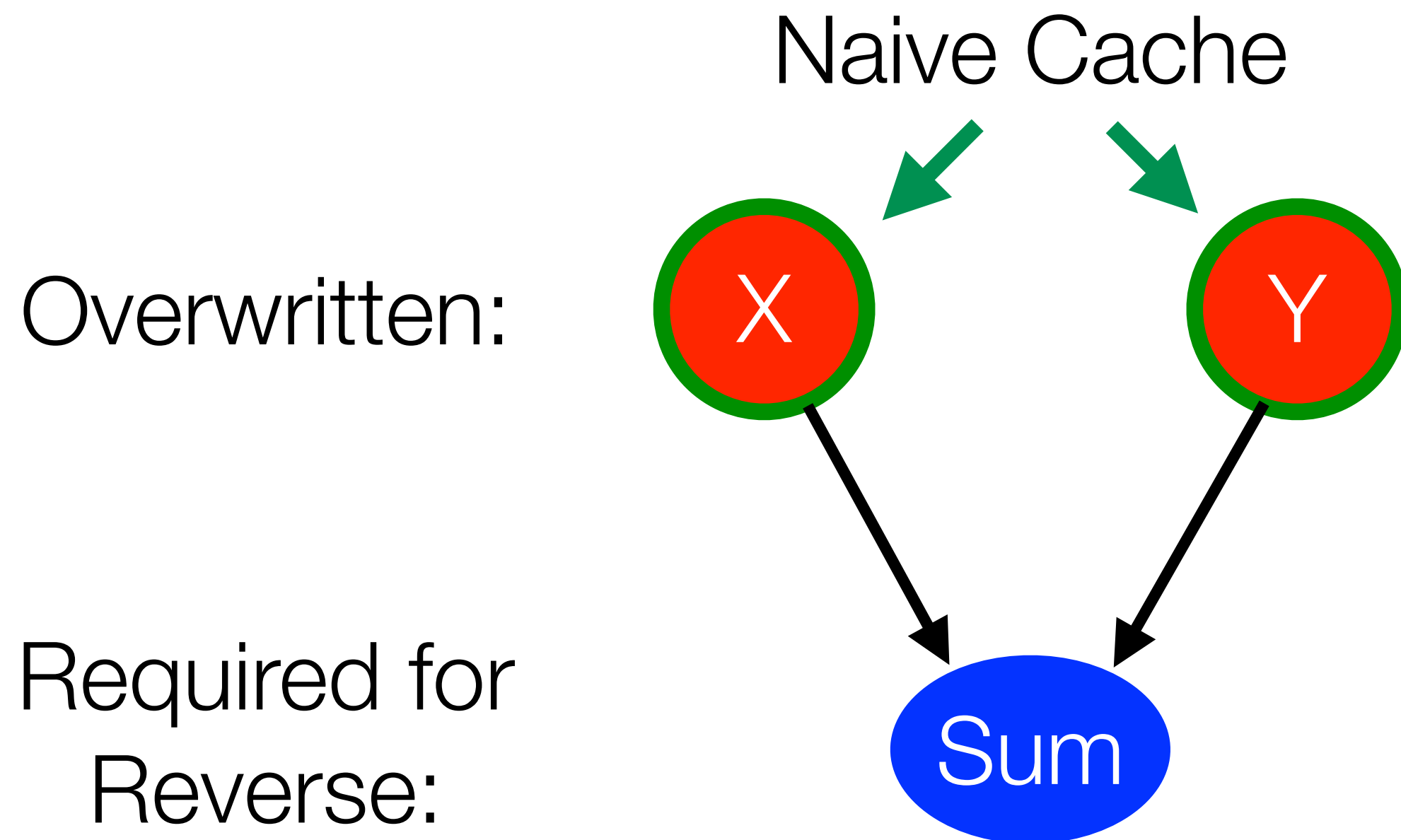
Required for  
Reverse:

```
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.

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

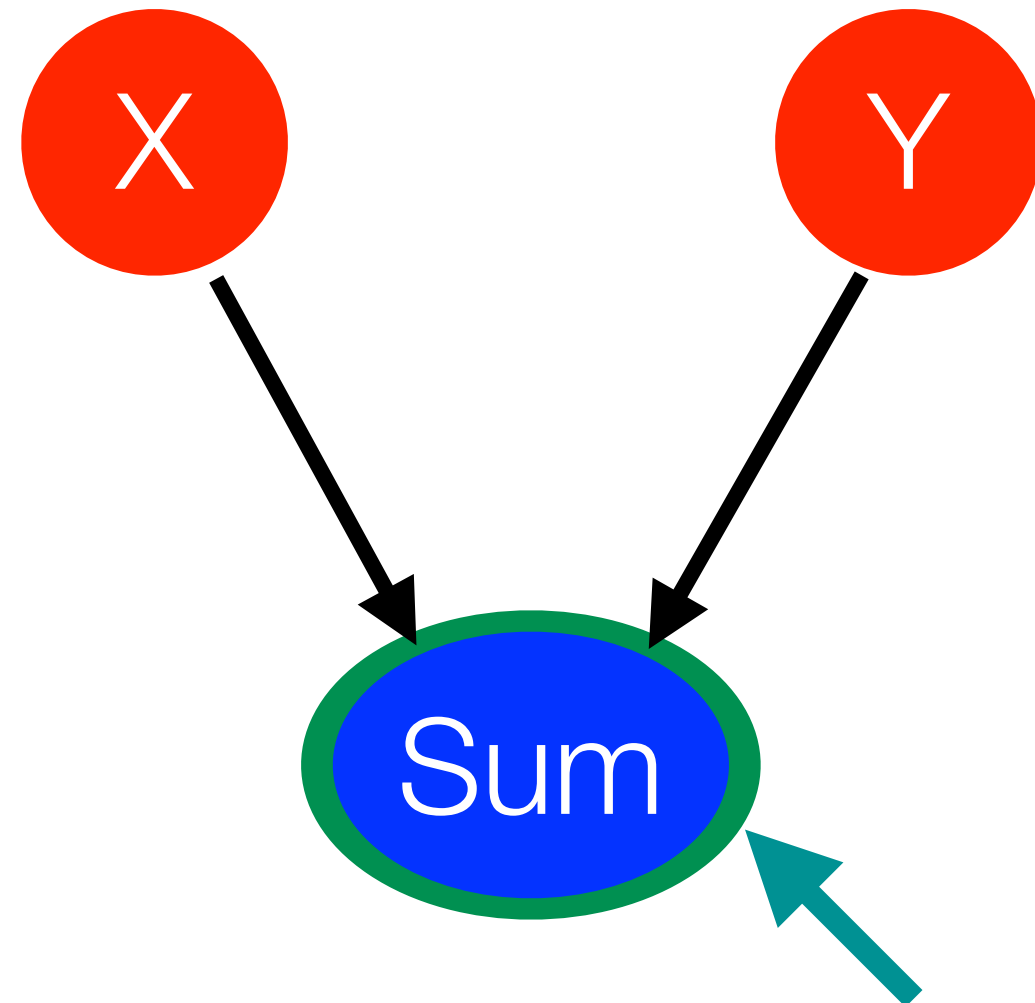
for(int i=0; i<10; i++) {
    double sum = x[i] + y[i];
    sum_cache[i] = sum;

    use(sum);
}

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

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

Overwritten:



Required for  
Reverse:

Smallest Cache

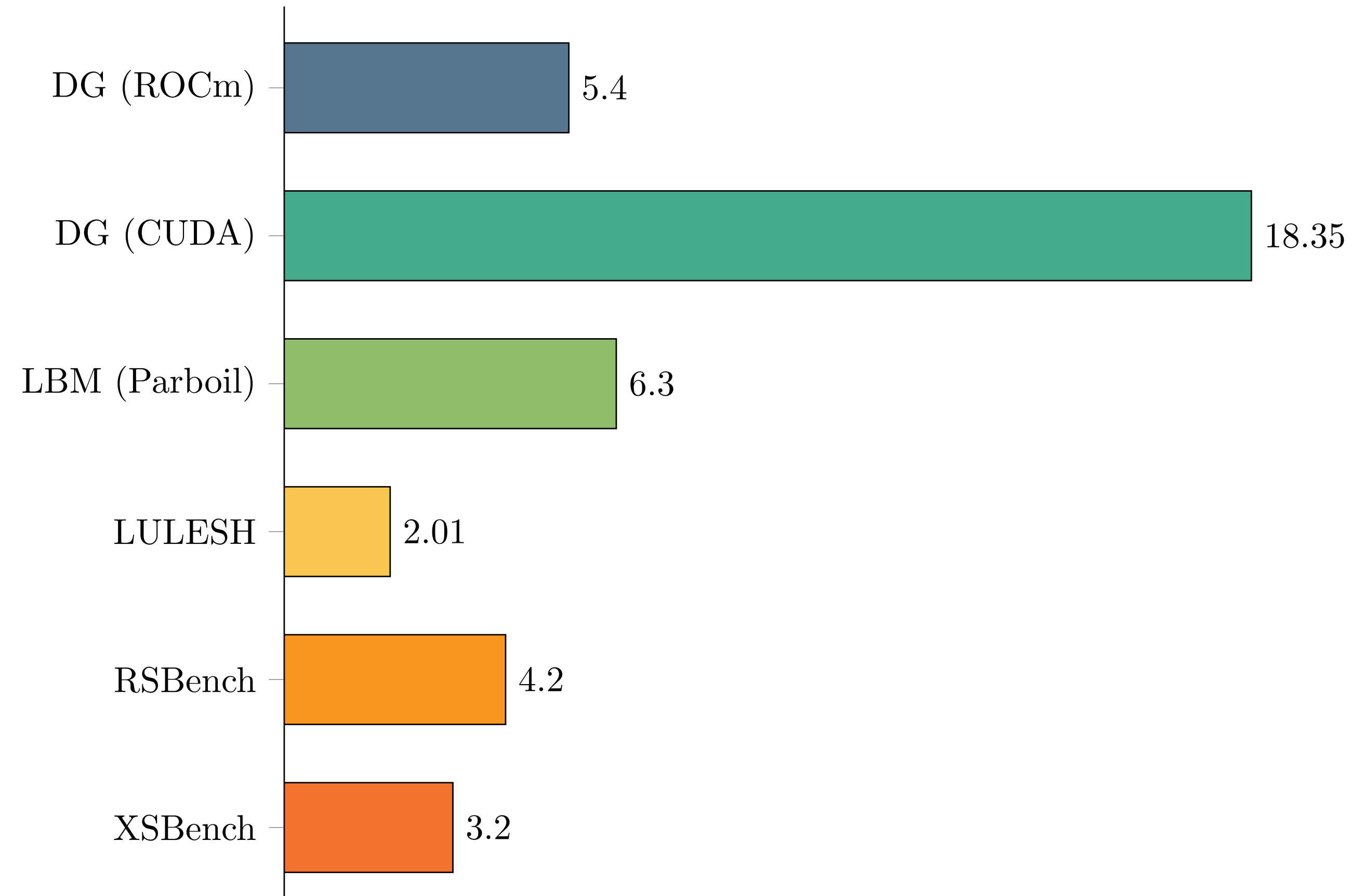
# Novel AD + GPU Optimizations

---

- See our SC'21 paper for more (<https://c.wsmoses.com/papers/EnzymeGPU.pdf>)  
Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. SC, 2021
- [AD] Cache LICM/CSE
- [AD] Min-Cut Cache Reduction
- [AD] Cache Forwarding
- [GPU] Merge Allocations
- [GPU] Heap-to-stack (and register)
- [GPU] Alias Analysis Properties of SyncThreads
- ...

# GPU Gradient Overhead

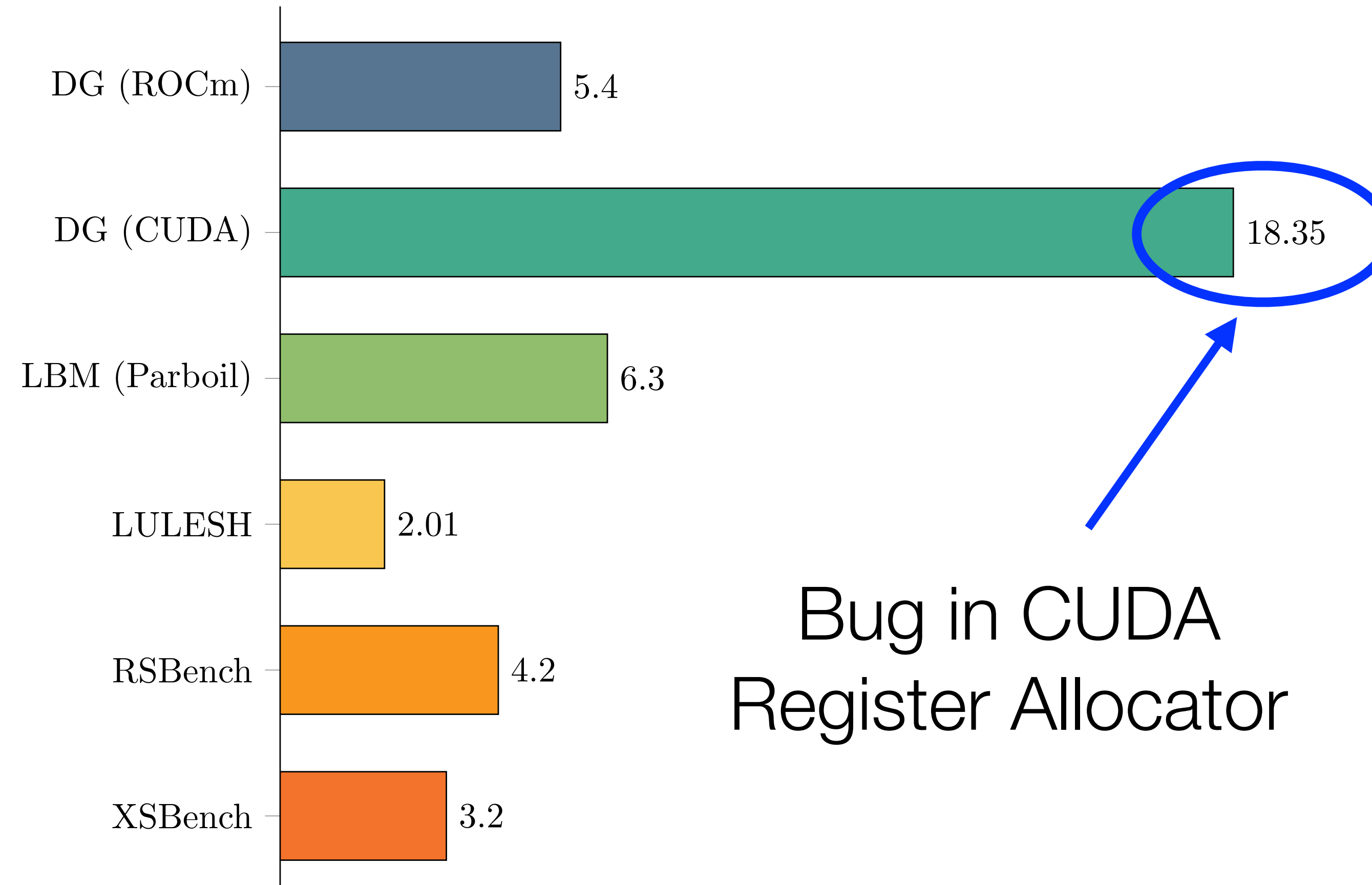
- Evaluation of both original code and gradient
  - DG: Discontinuous-Galerkin integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute bound, respectively)





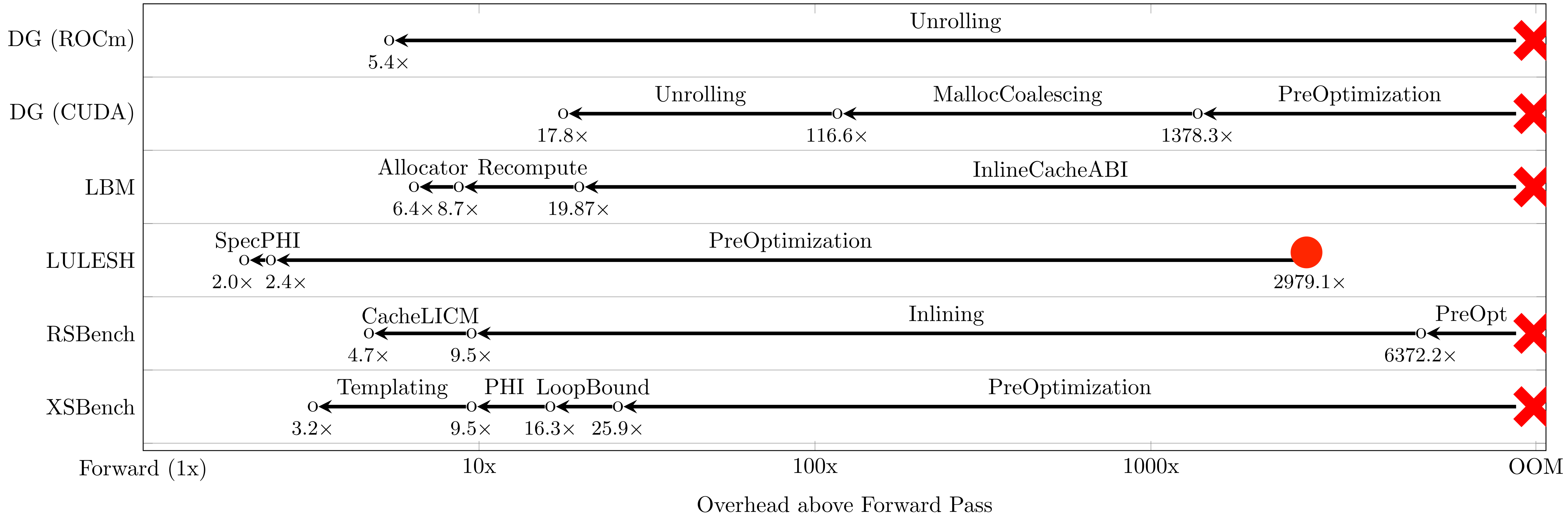
# GPU Gradient Overhead

- Evaluation of both original code and gradient
  - DG: Discontinuous-Galerkin integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute bound, respectively)

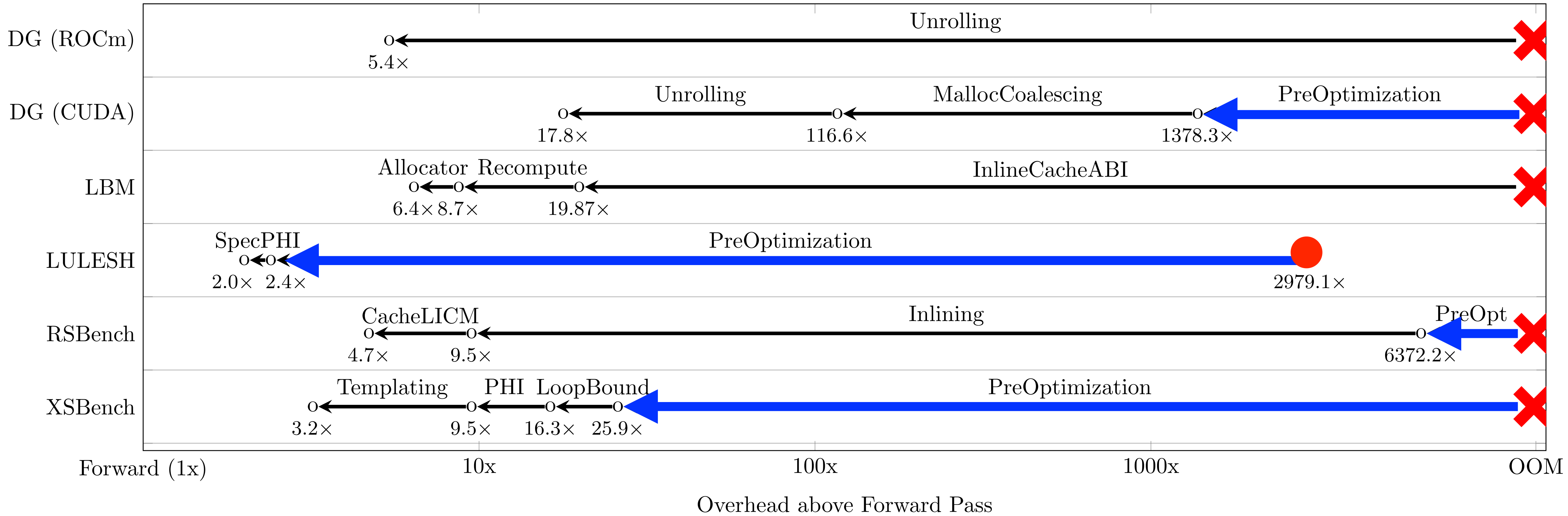


Bug in CUDA  
Register Allocator

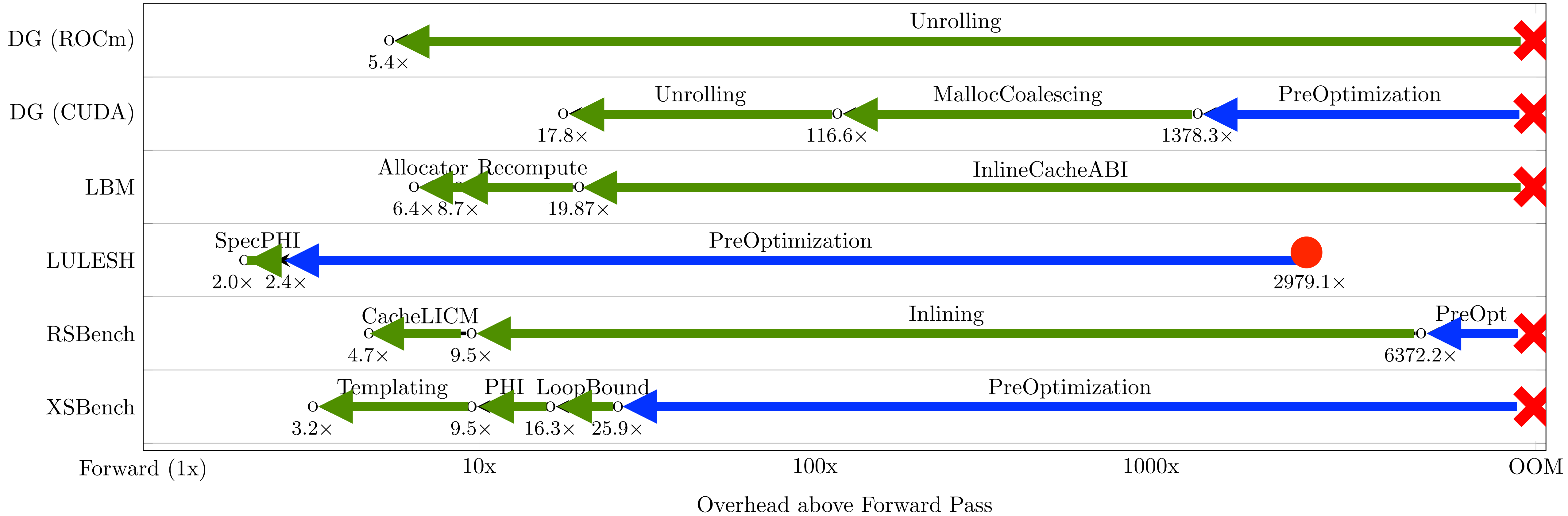
# Ablation Analysis of Optimizations



# Ablation Analysis of Optimizations

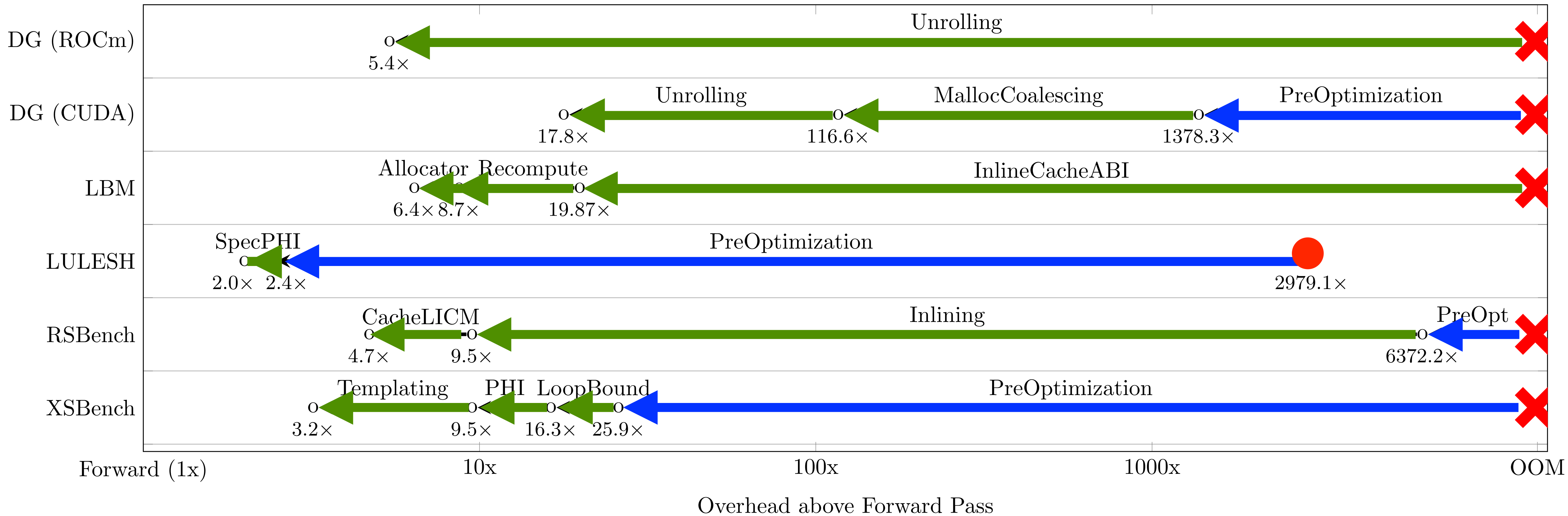


# Ablation Analysis of Optimizations





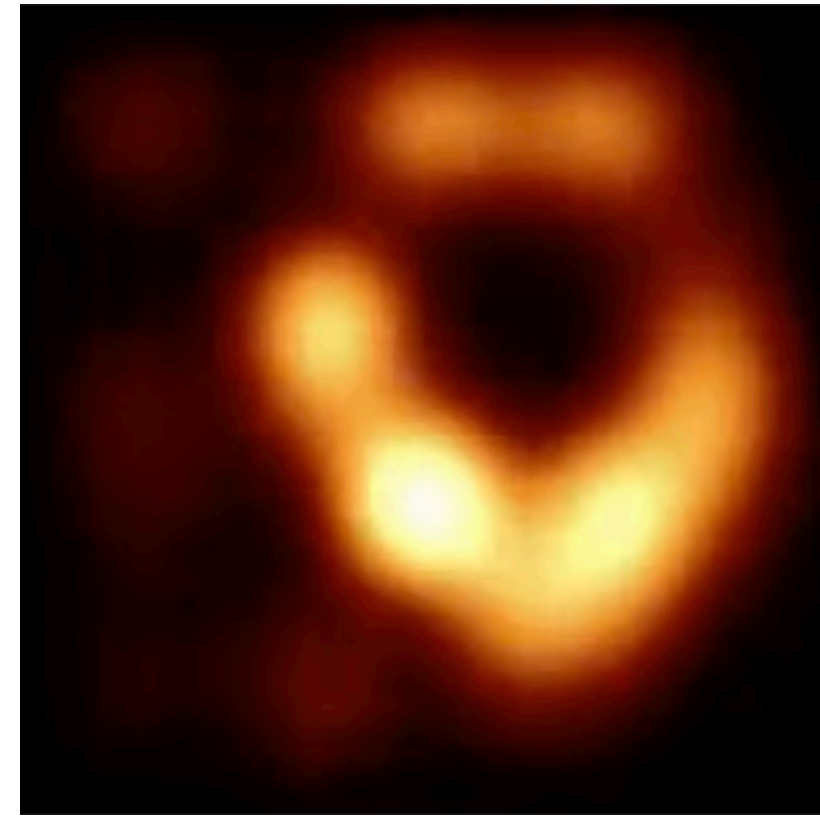
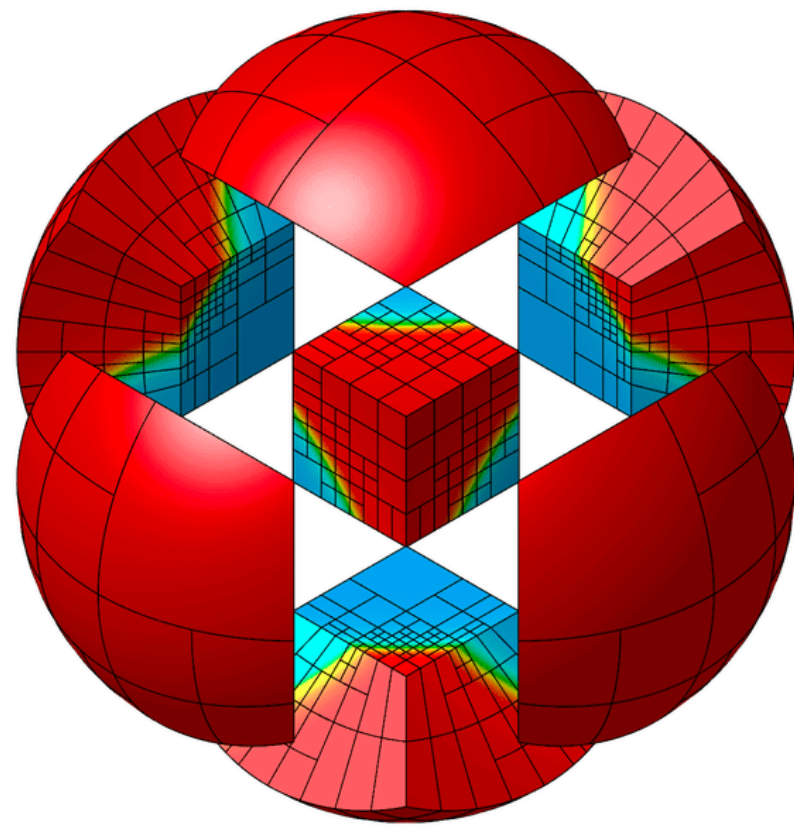
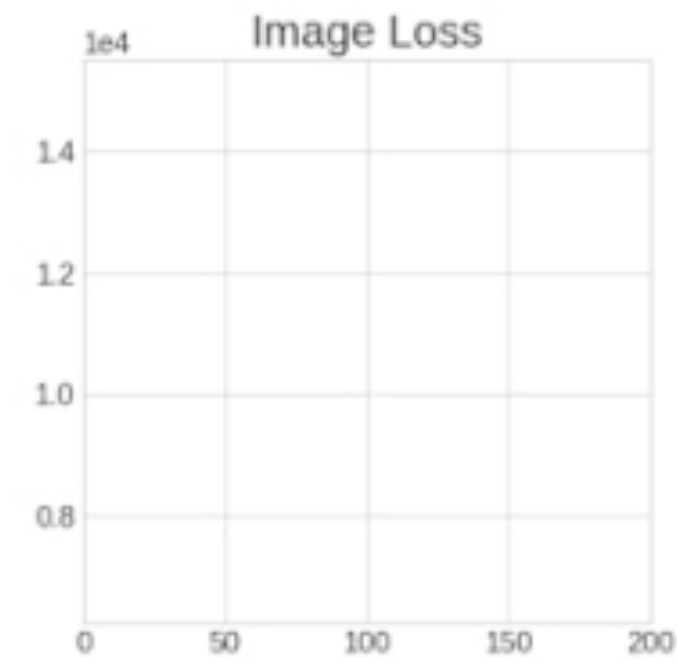
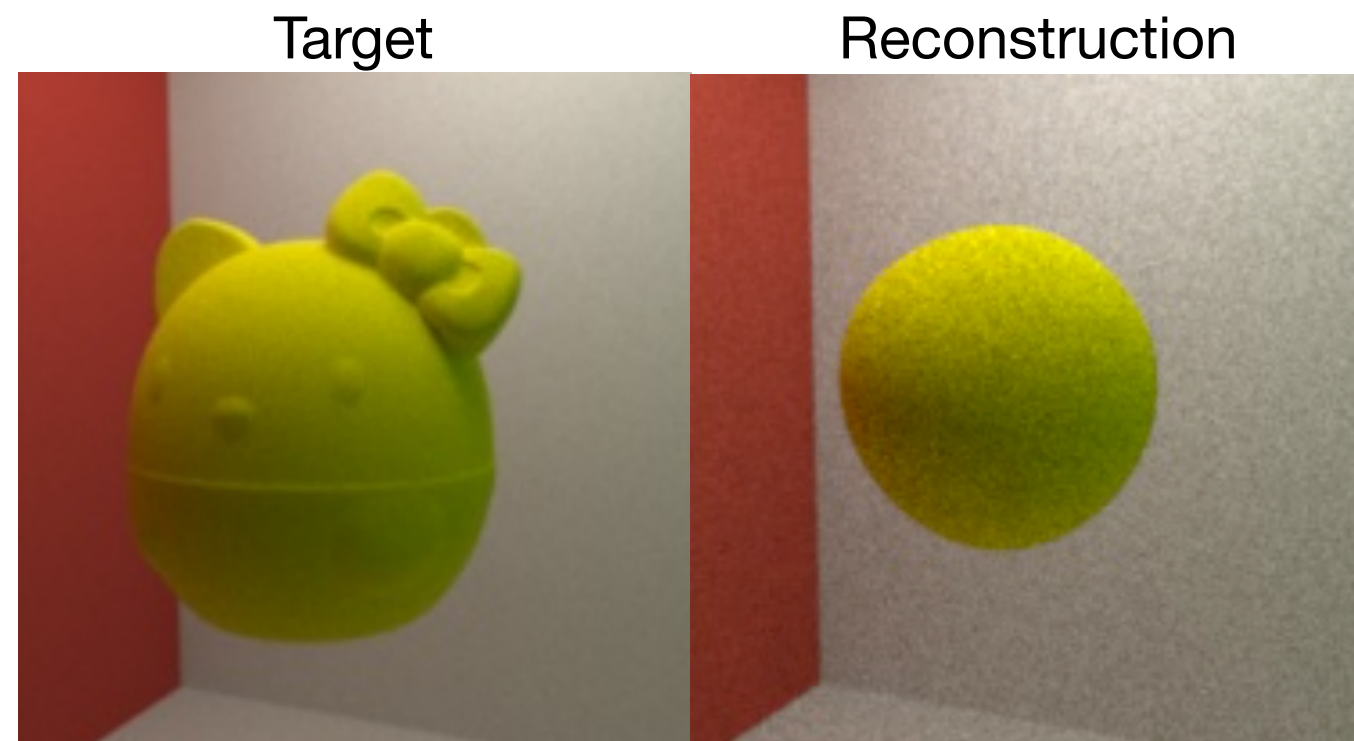
# Ablation Analysis of Optimizations



GPU AD is Intractable Without Optimization!



# Enzyme-Powered Applications



**>100x speedup!**  
Prior:  
**5 days (cluster)**  
Enzyme-Based:  
**1 hour (laptop)**

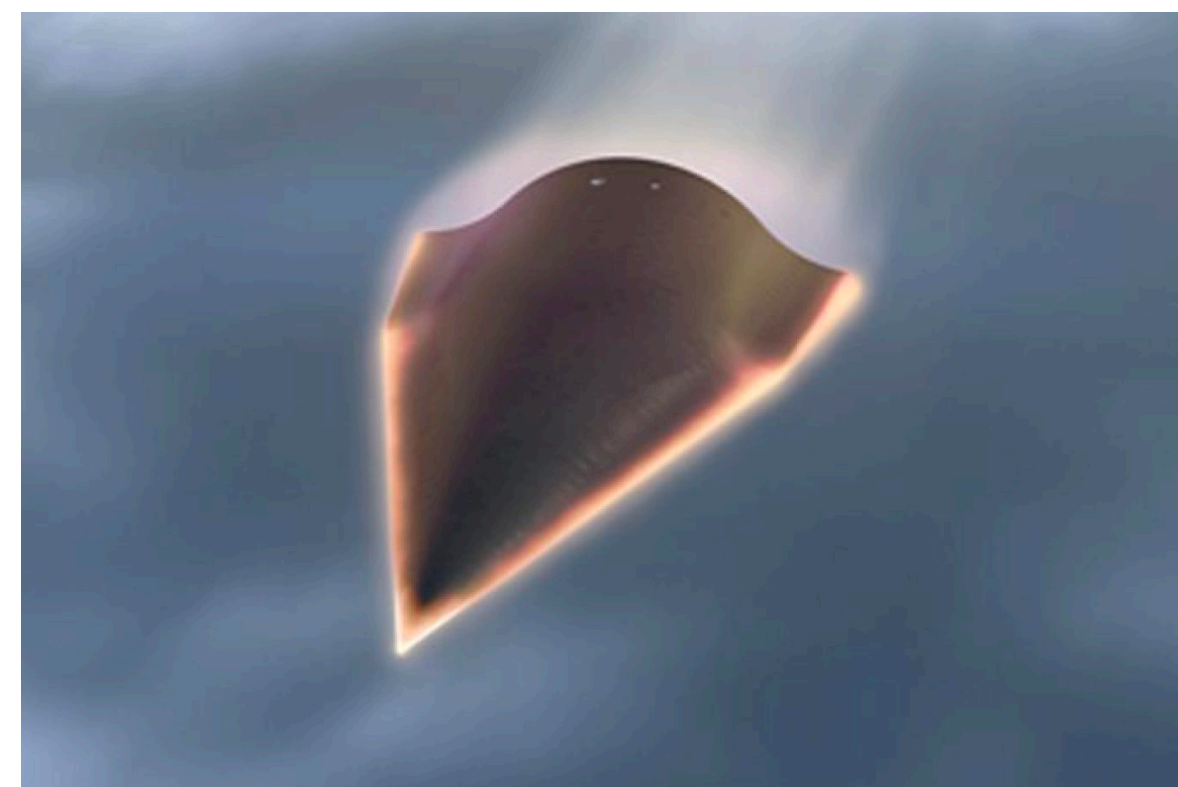
from [Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering](#), SIGGRAPH Asia 2022, Zihan Yu et al

from [MFEM Team at LLNL](#)

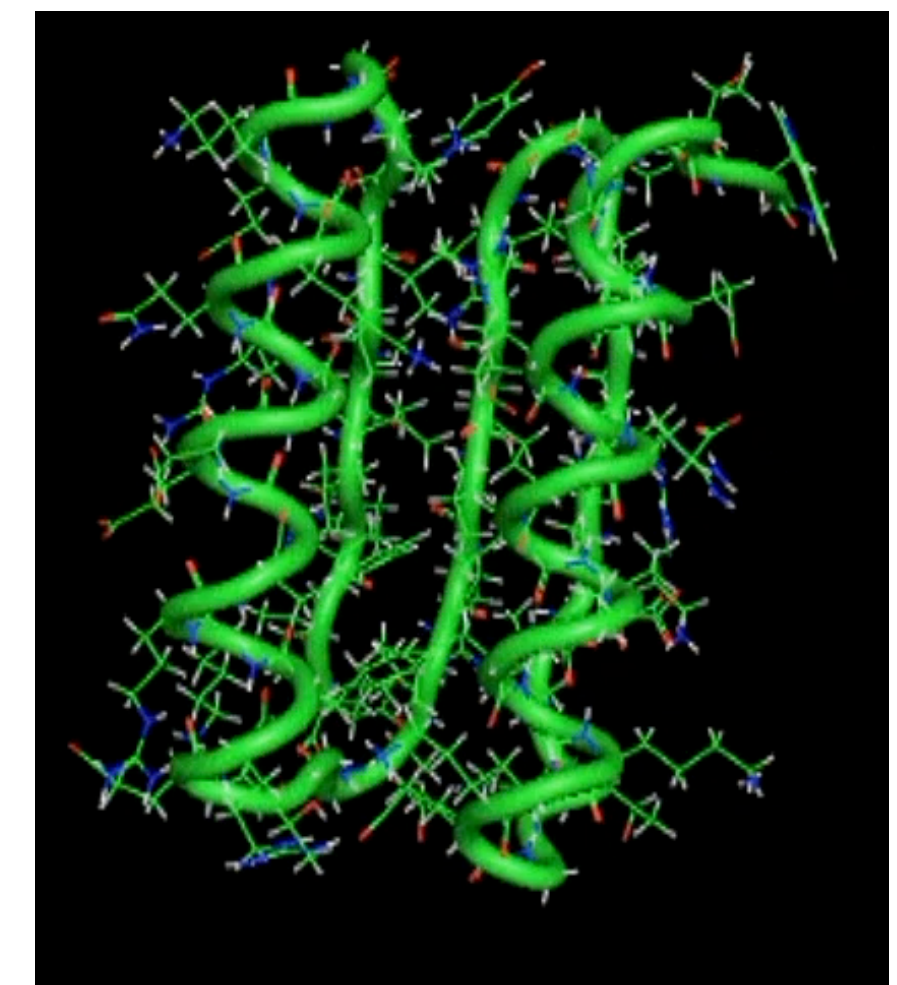
from [Comrade: High Performance Black-Hole Imaging](#) JuliaCon 2022, Paul Tiede (Harvard)



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



from [Center for the Exascale Simulation of Materials in Extreme Environments](#)



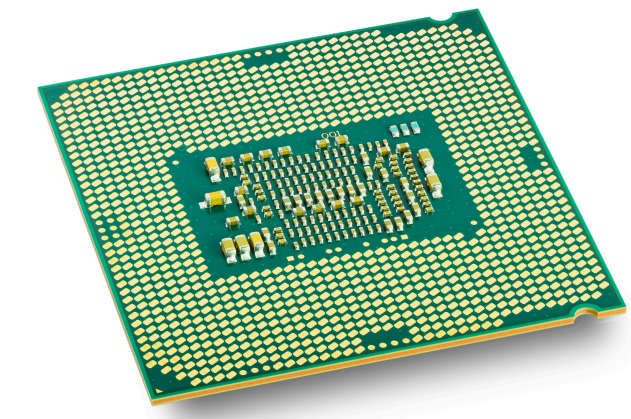
from [Differential Molecular Simulation with Molly.jl](#), EnzymeCon 2023, Joe Greener (Cambridge)



# The HPC Landscape Today

---

- Cutting-edge scientific computing requires efficiently leveraging *parallelism*
  - Multicore chips
  - Distributed clusters
  - Accelerators (e.g. GPUs, TPUs)



# Case Study: Parallel 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);
    }
}
```

N = 64M



# Case Study: Parallel 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);
    }
}
```

N = 64M

Serial Running time: 0.312 s

# Case Study: Parallel Vector Normalization

---

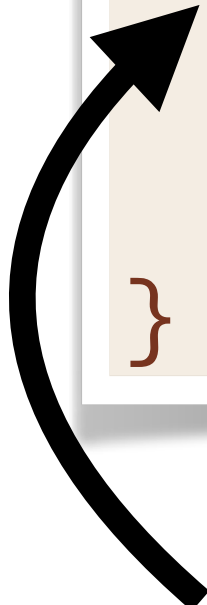
```
//Compute magnitude in O(n)
double mag(double[] x);

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

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

N = 64M

Serial Running time: 0.312 s



A parallel loop replaces  
the original serial loop

# Case Study: Parallel Vector Normalization

---

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

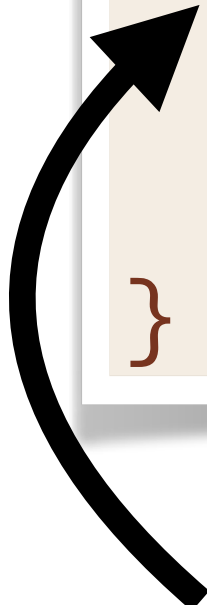
//Compute norm in O(n^2) work
void norm(double[] out, double[] in) {

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

N = 64M

Serial Running time: 0.312 s

18-core Running time: 180.657s



A parallel loop replaces  
the original serial loop

# Case Study: Parallel Vector Normalization

---

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

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

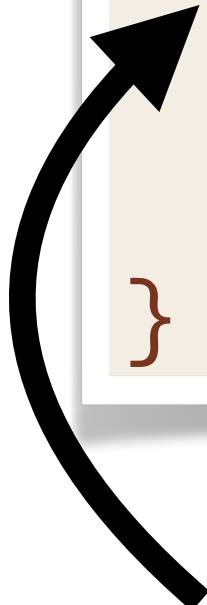
    parallel_for (int i=0; i<n; i++) {
        out[i] = in[i] / mag(in);
    }
}
```

N = 64M

Serial Running time: 0.312 s

18-core Running time: 180.657s

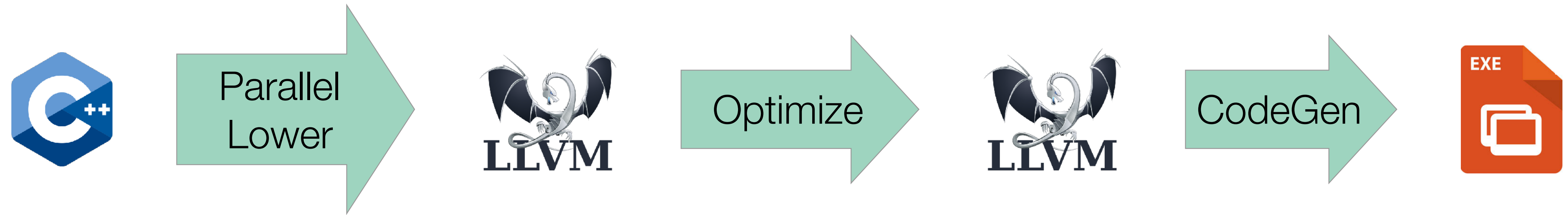
1-core Running time: 2600.287s



A parallel loop replaces  
the original serial loop

# Why the Parallel Slowdown?

---



Frontend directly translates parallel language constructs

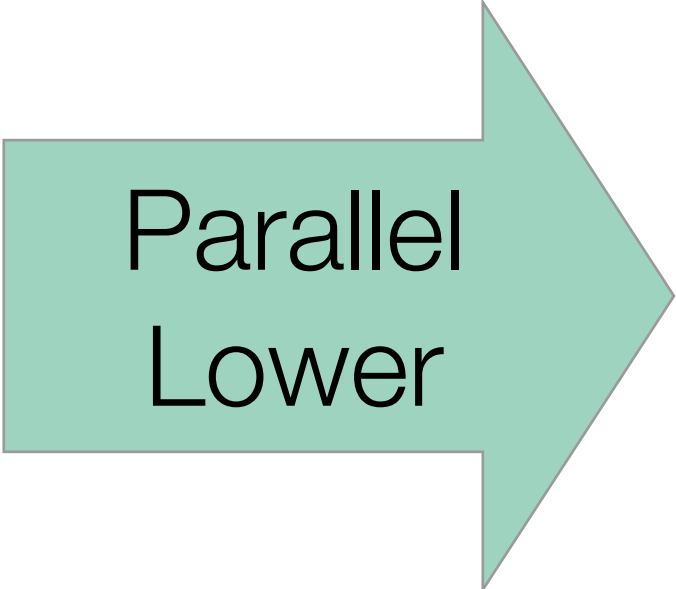


# Compiling Parallel Code

---

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

Parallel  
Lower



```
void norm(double[] out, double[] in)
{
    struct args_t args = { out, in };
    __cilkrts_pfor(body, args, 0, n);
}

void body(struct args_t args, int i)
{
    double *out = args.out;
    double *in = args.in;
    out[i] = in[i] / mag(in);
}
```

# Compiling Parallel Code

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

Parallel  
Lower

```
void norm(double[] out, double[] in)
{
    struct args_t args = { out, in };
    __cilkrts_pfor(body, args, 0, n);
}

void body(struct args_t args, int i)
{
    double *out = args.out;
    double *in = args.in;
    out[i] = in[i] / mag(in);
}
```

The compiler doesn't understand the parallel runtime and cannot move mag

# Compiling Parallel Code (Realistic)

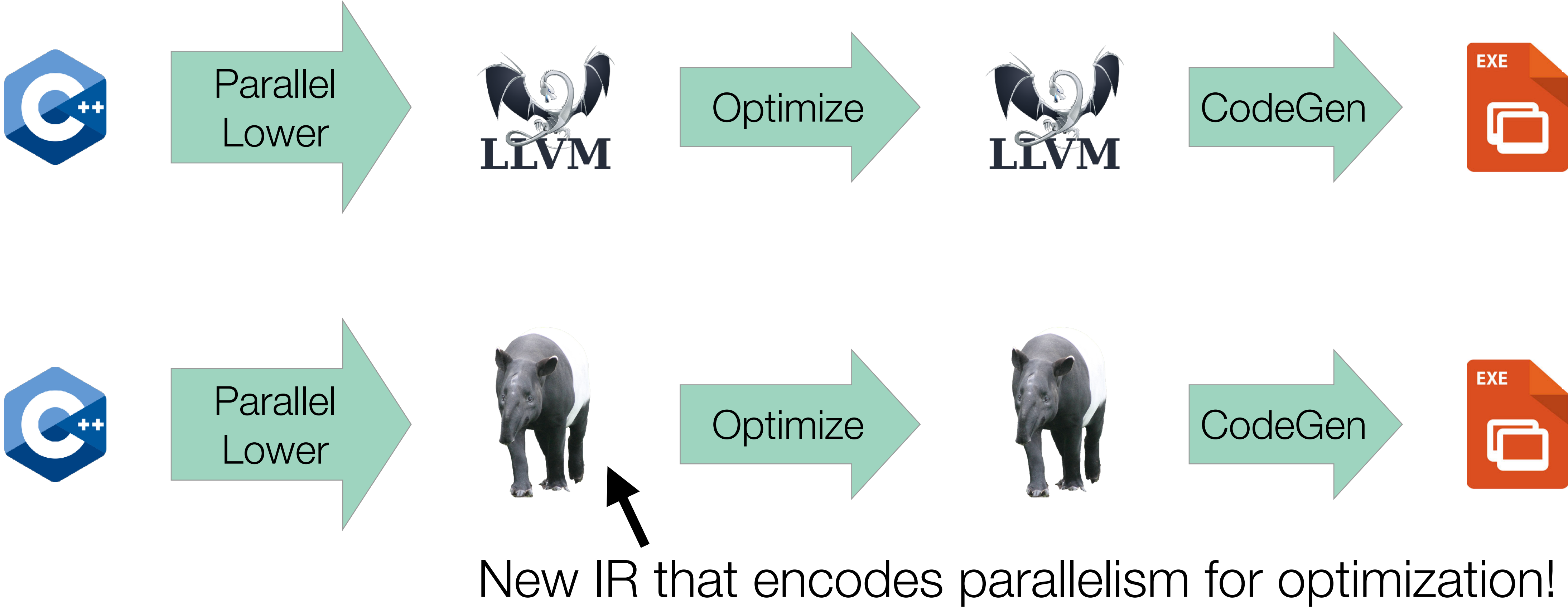
```
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = spawn fib(n - 1);
    y = fib(n - 2);
    sync;
    return x + y;
}
```

Parallel  
Lower

```
int fib(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    if (n < 2) return n;
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_fib(&x, n-1);
    y = fib(n-2);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
    return result;
}

void spawn_fib(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = fib(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```

# Idea: New Parallel Compilation Pipeline



# Parallel IR: A Bad Idea?

---

From “[LLVMdev] LLVM Parallel IR,” 2015:

- “[I]ntroducing [parallelism] into a so far ‘sequential’ IR will cause **severe breakage and headaches.**”
- “[P]arallelism is invasive by nature and would have to **influence most optimizations.**”

Other communications, 2016–2017:

- “There are **a lot of information needs** to be represented in IR for [back end] transformations for OpenMP.” [Private communication]
- “If you support all [parallel programming features] in the IR, **a \*lot\* [of LOC]...would probably have to be modified** in LLVM.” [[RFC] IR-level Region Annotations]



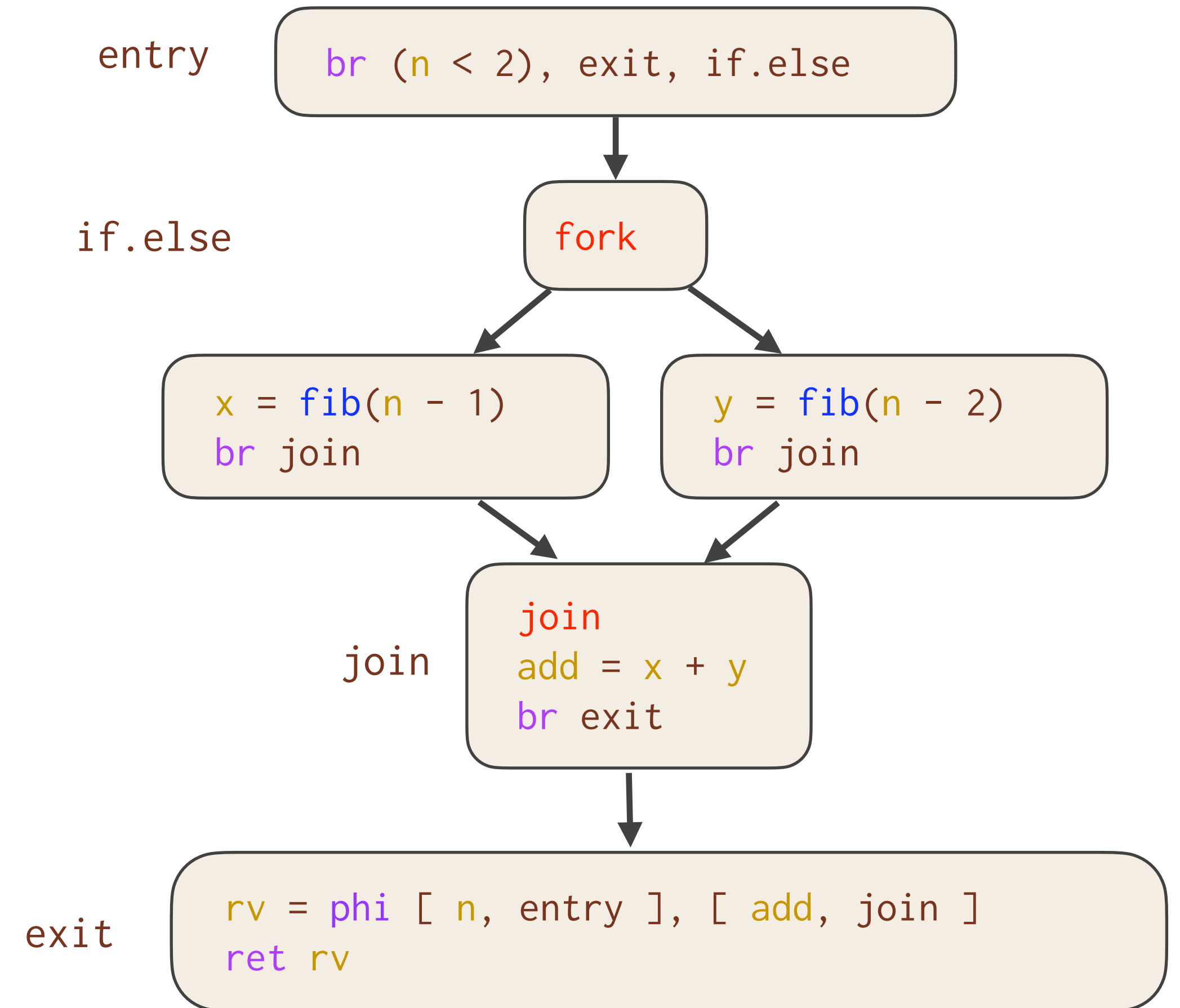
# Example Previous Parallel IR

- Previous CFG-based parallel IR's represented tasks **symmetrically**.

```
int fib(int n) {  
  if (n < 2) return n;  
  int x, y;  
  x = spawn fib(n - 1);  
  y = fib(n - 2);  
  sync;  
  return x + y;  
}
```

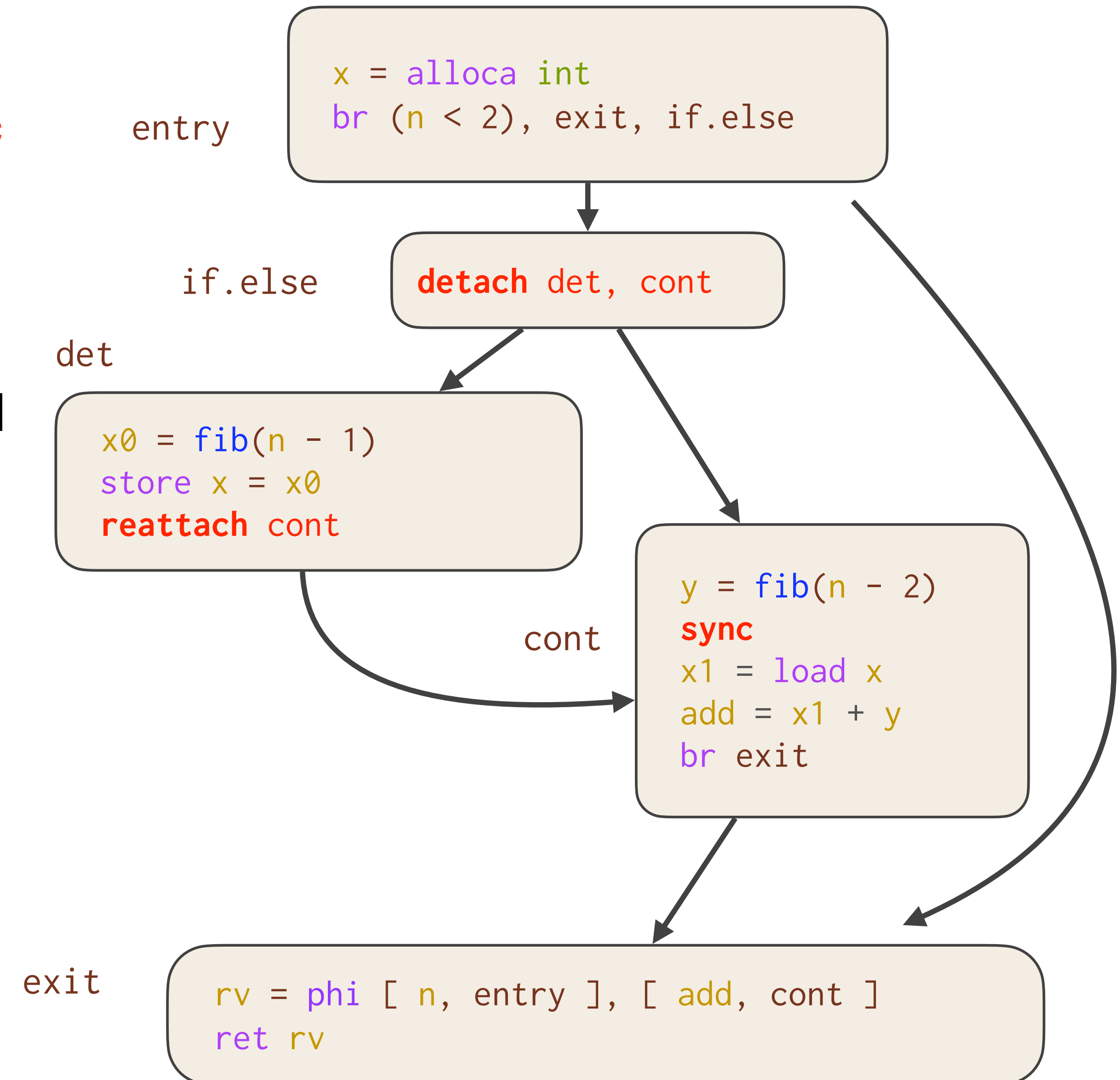
**Problem:** The join block **breaks implicit assumptions** made by the compiler.

**Example:** Values from **all** predecessors of a join must be available at runtime [LMP97].



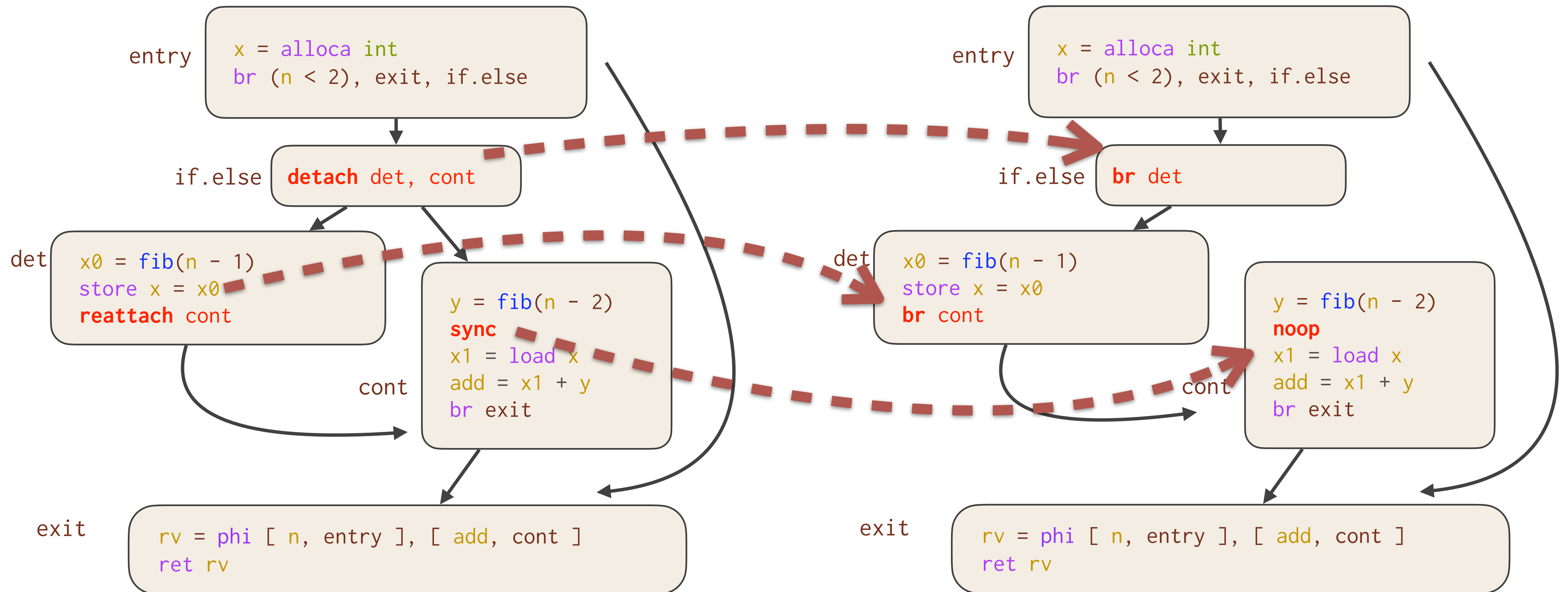
# Tapir: Task-Based Asymmetric Parallel IR

- Tapir models parallel tasks **asymmetrically** via three new instructions: **detach**, **reattach**, and **sync**
- The successors of a detach **may** run in parallel.
- Code after a **sync** is guaranteed to have completed previously detached tasks.
- Tapir simultaneously represents the **serial** and **parallel** semantics of the program.



# Tapir: Task-Based Asymmetric Parallel IR

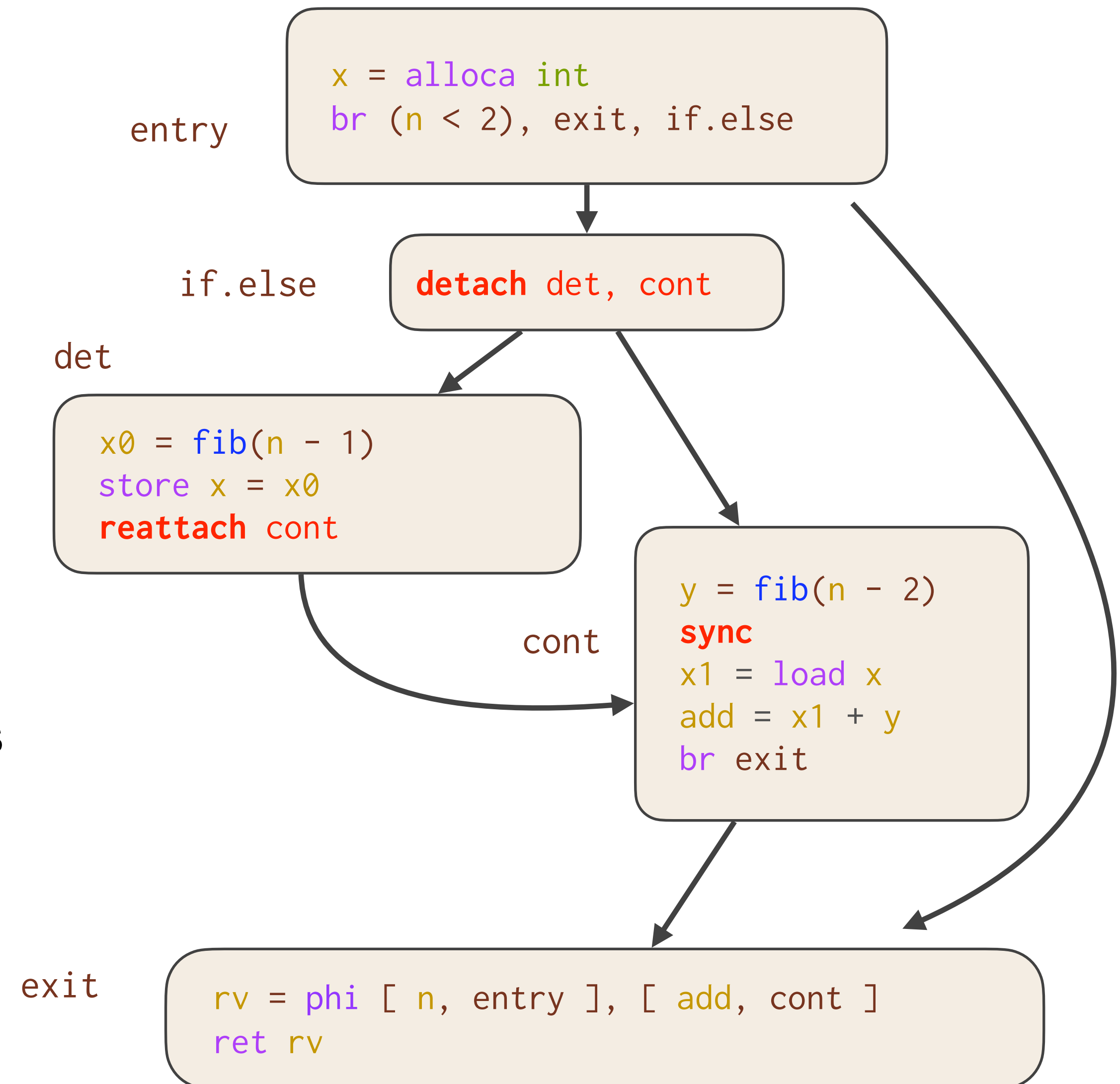
- Reasoning about parallelism is a minor change to reasoning about the **serial projection**.



# Maintaining Correctness

**Problem:** How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- Consider moving memory operations around each new instruction.
- Moving code above a **detach** or below a **sync** serializes it and is always valid.
- Other potential races are handled by giving **detach**, **reattach**, and **sync** appropriate attributes and by slight modifications to **mem2reg**.

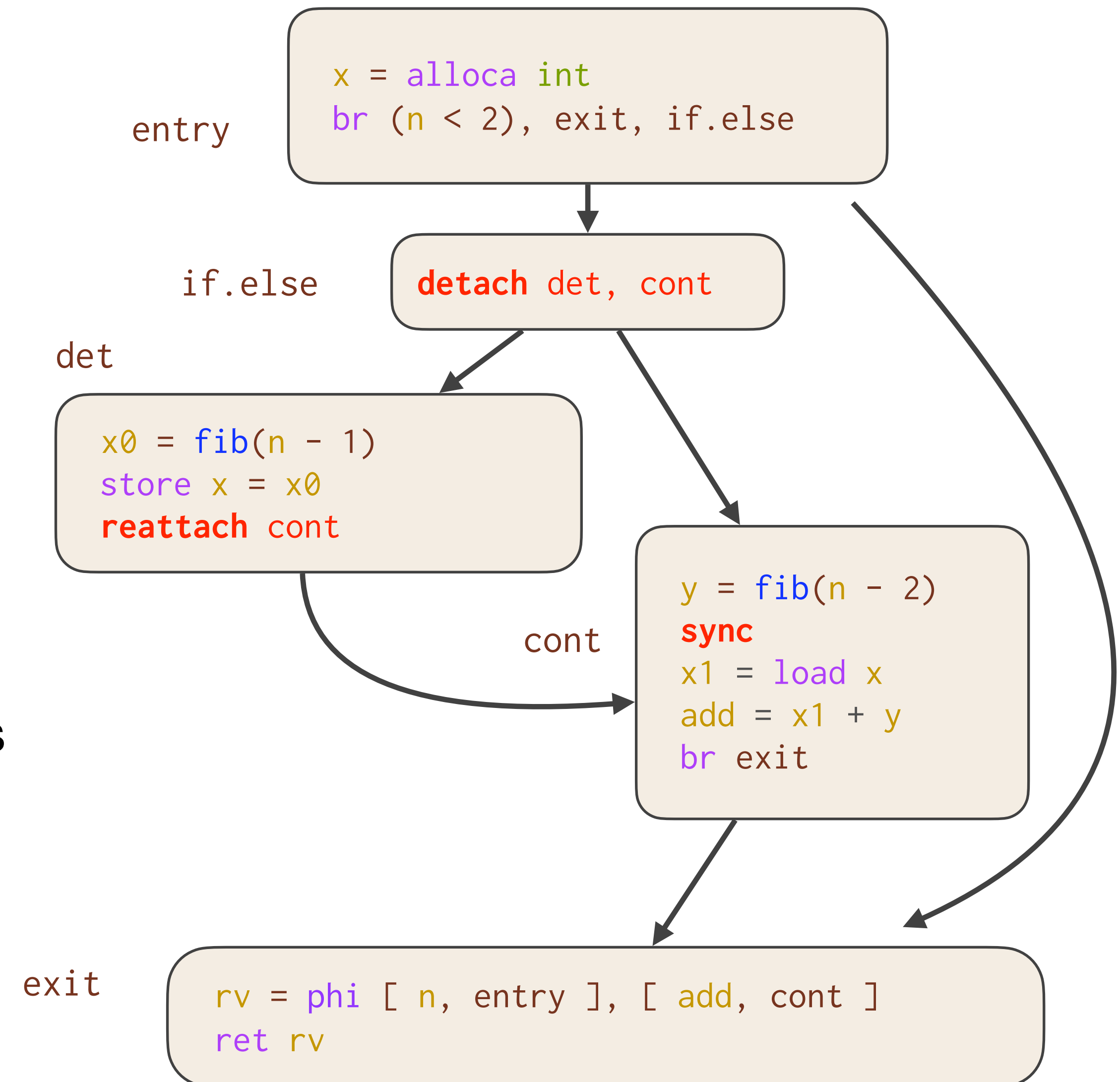


# Maintaining Correctness

**Problem:** How does the compiler ensure that code motion does not introduce a determinacy race into otherwise race-free code?

- Consider moving memory operations around each new instruction.
- Moving code above a **detach** or below a **sync** serializes it and is always valid.
- Other potential races are handled by giving **detach**, **reattach**, and **sync** appropriate attributes and by slight modifications to **mem2reg**.

Serial optimization passes  
do not create bugs!







# Vector Normalization with a Parallel-Aware Compiler

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

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

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

A parallel loop replaces  
the original serial loop

N = 64M

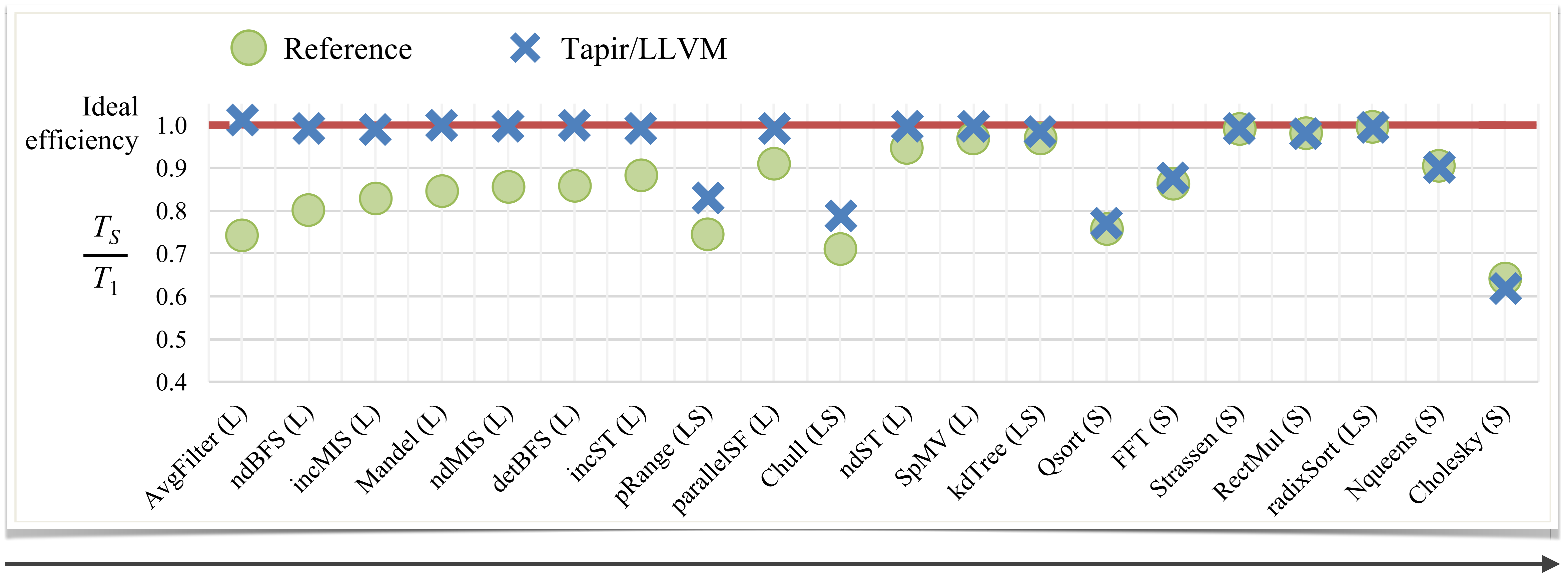
|                       |         |
|-----------------------|---------|
| Serial Running time:  | 0.312 s |
| 18-core Running time: | 0.081 s |
| 1-core Running time:  | 0.321 s |

Great work efficiency!

$$T_S / T_1 = 97\%$$

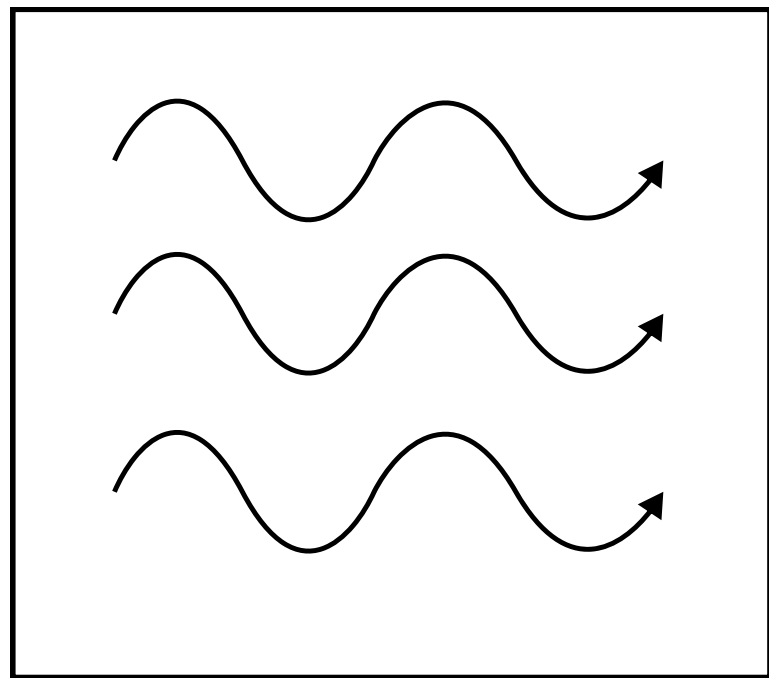


# Vector Normalization with a Parallel-Aware Compiler



Decreasing difference between Tapir/LLVM and Reference

# Revisiting The Programmer's Burden



```

Node 1 float y = f(x);
      MPI_Send(&y, ...);

Node 2 float y;
      MPI_Recv(&y, ...);

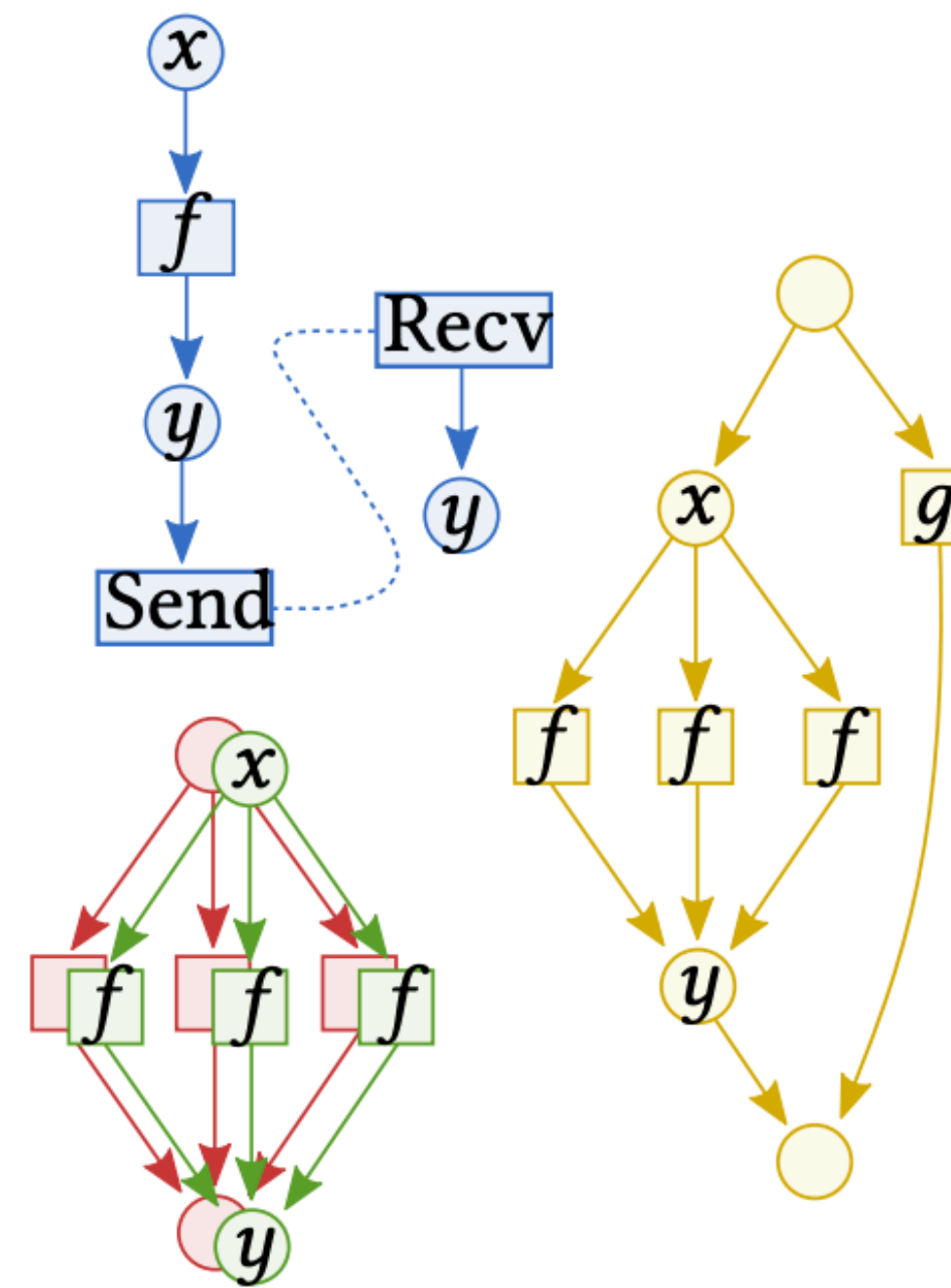
#pragma omp parallel for
for (int i=0; i<3; ++i){
  y[i] = f(x[i]);
}

Threads.@threads for i=1:3
  y[i] = f(x[i])
end

@sync begin
  @spawn @sync for i in i:3
    @spawn f(x(i))
  end
  @spawn g()
end
    
```

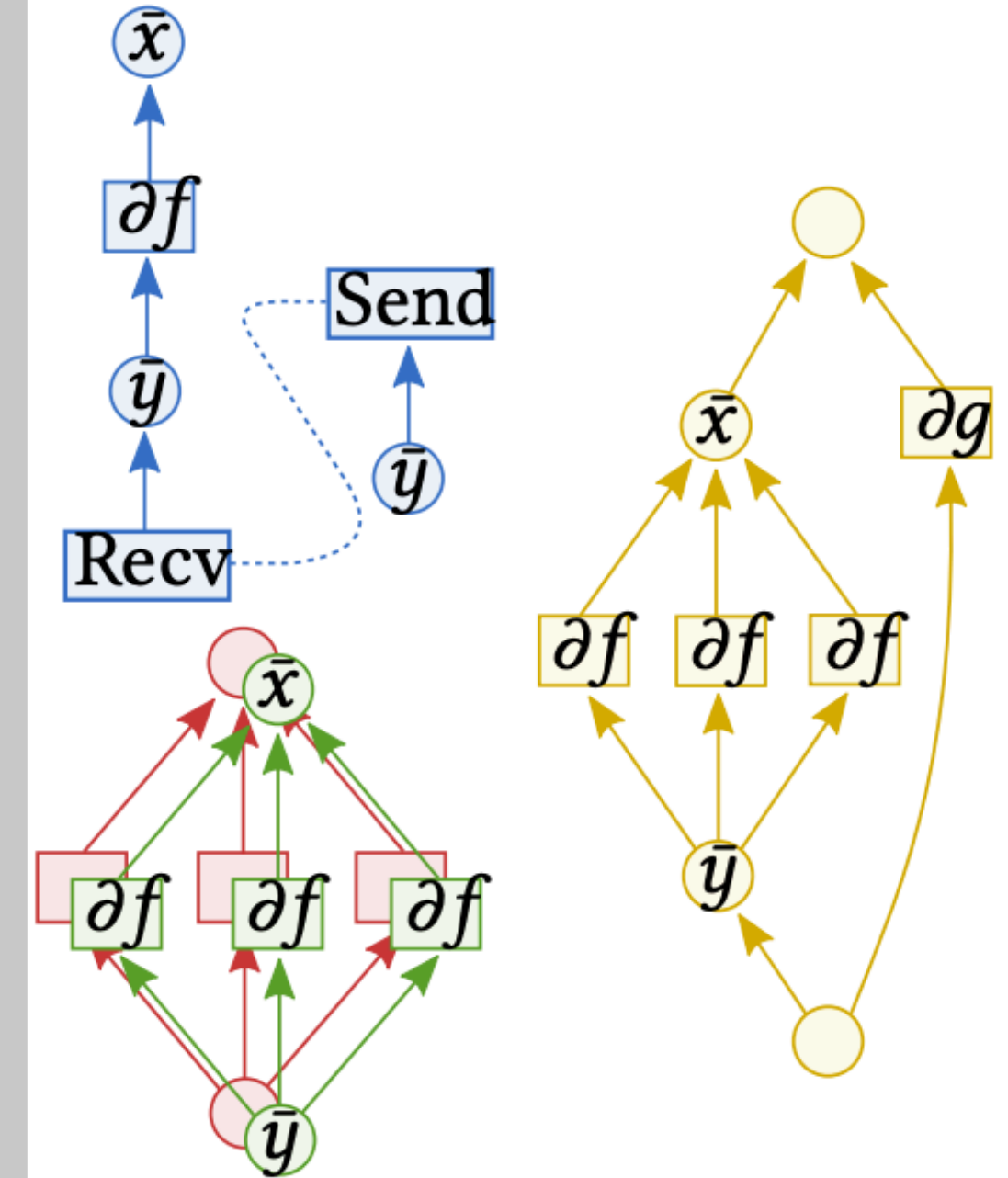
Code

Compiler



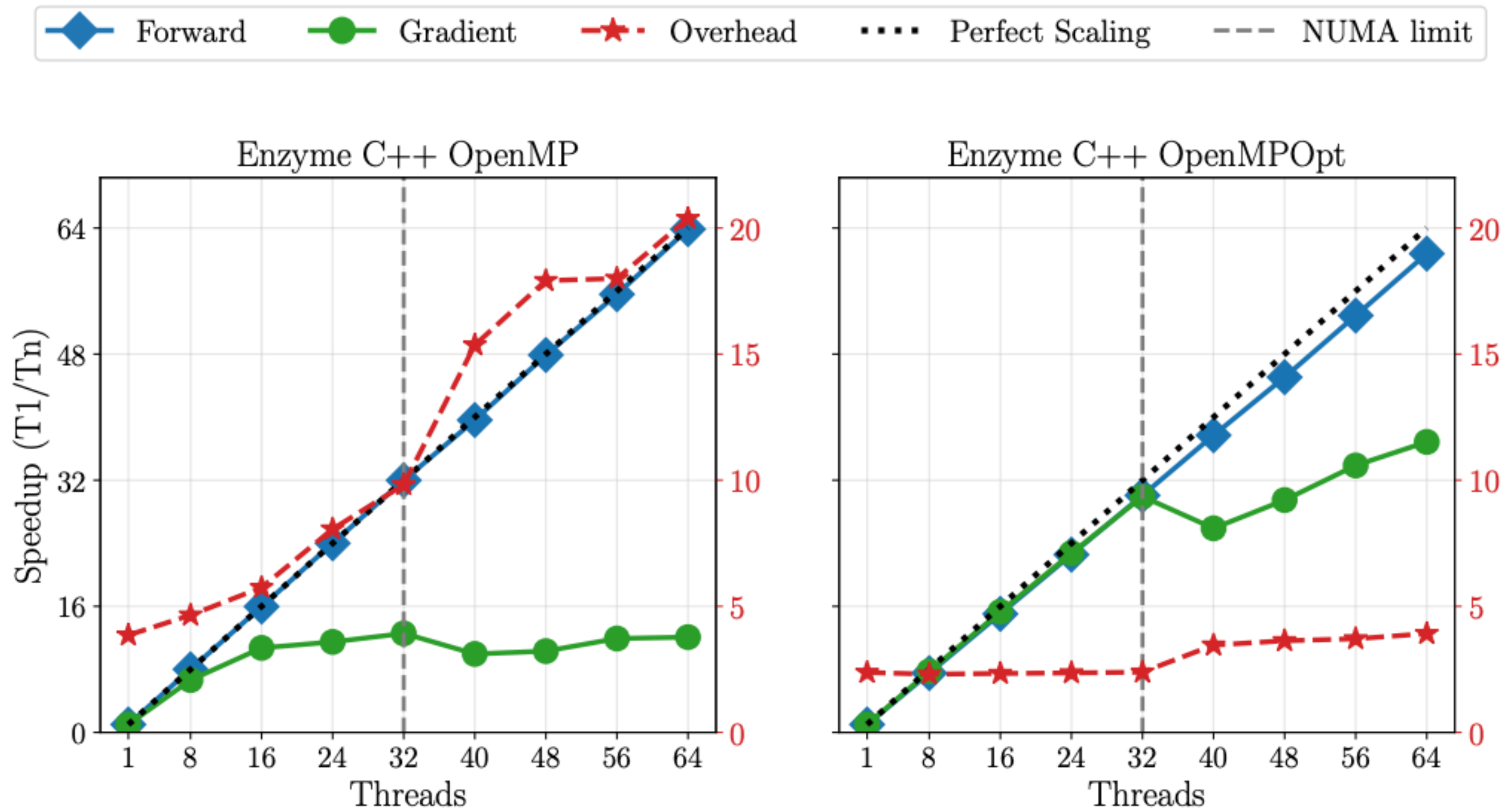
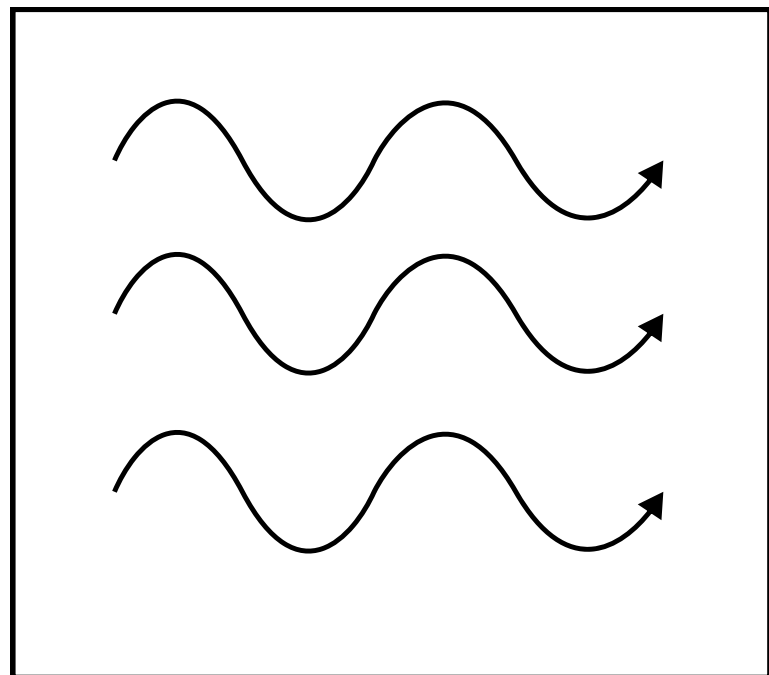
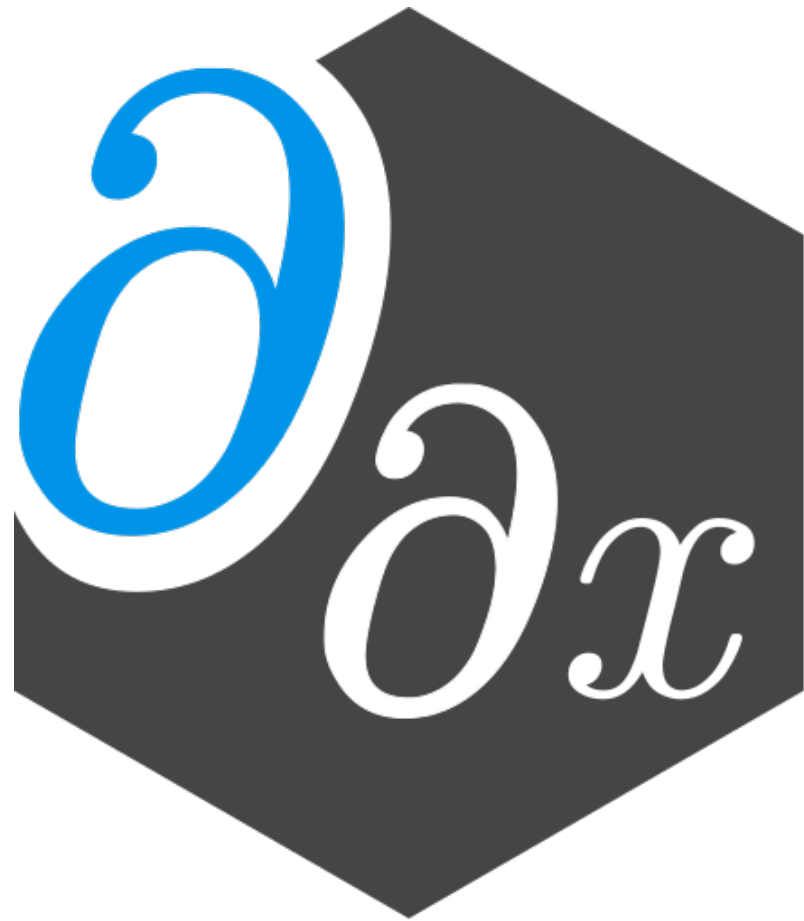
Compiler IR

Enzyme



Reverse-Mode  
Derivative

# Revisiting The Programmer's Burden (published at SC22)







# Conclusions

---

- Explosion of specialized software packages and hardware architectures -> scientists spending more time learning how to optimize programs and use platform-specific API's than working on their intended problem.
- Rather than burdening the user, compilers can automatically generate fast, portable, and composable code.

 Enzyme generates fast derivatives of programs needed for science and machine learning, *without user rewriting*

 Tapir understands the parallelism within programs, enabling existing optimizations to apply with minimal modification.

- All these tools are open source and used in academia and industry and in disciplines that range from climate science to physics to material science

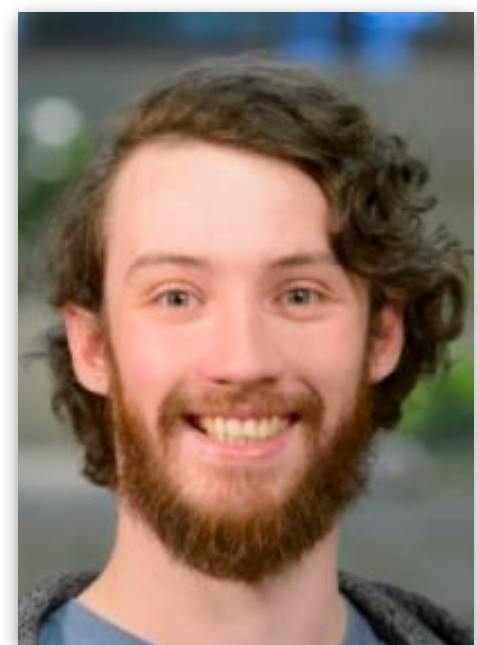
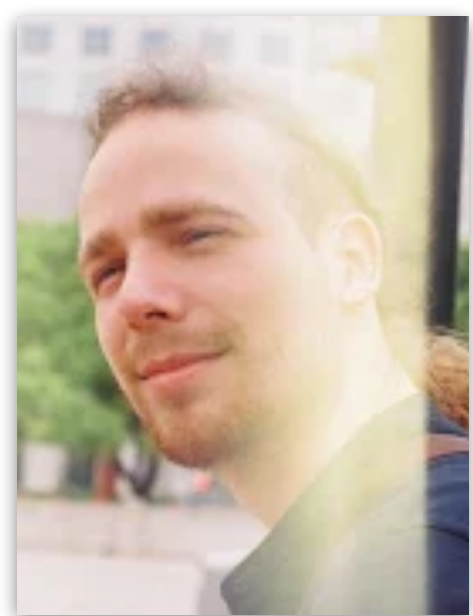


# Acknowledgements

---

- Thanks to my family for supporting me, including Marina Moses, John Moses, Sophia Moses, and Panayoti Stefanidis.
- Many thanks to so many colleagues for help with this work including: Srini Devadas, James Bradbury, Jed Brown, Alex Chernyakhovsky, Valentin Churavy, Lilly Chin, Hal Finkel, Marco Foco, Leila Gharaffi, Laurent Hascoet, Patrick Heimback, Paul Hovland, Jan Hueckelheim, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Ludger Paehler, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Vassil Vassilev, Sarah Williamson, Alex Zinenko, Pat McCormick, George Stelle, Stephen Olivier, Joanna Balme, Eric Brown-Dymkosky, Victor Guerrero, Stephen Jones, Andre Kessler, Adam Lichtl, Kevin Lung, Ken Museth, Nathan Robertson, Youseef Marzouk, Kevin Sabo, Jesse Michel, Cat Zeng, Allison Tam, Kevin Kwok, Will Bradbury, Alex Atanasov, Joe Murphy, Jamie Voros, Logan Engstrom, Douglas Kogut, Jiahao Li, Bojan Serafimov, Carl Guo, Sanath Govindarajan, Walden Yan, Sage Simhon, Chuyang Chen, Shakil Ahmed, Abhishek Vu, Chris Hill, Chris Peterson, Emma Batson, & more.
- Thank you to all my friends from MIT, TJ, NOVA, and beyond.





Valentin Churavy

Leila Ghaffari

Ludger Paehler

Johannes  
Doerfert

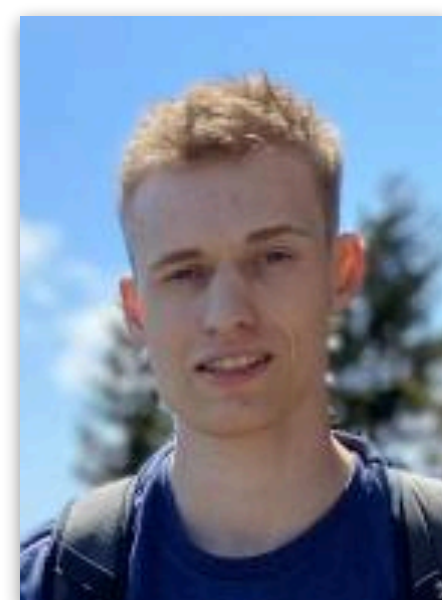
Jan Hückelheim

Charles E.  
Leiserson

Zach Devito

Andrew Adams

Lorenzo  
Chelini



Sri Hari Krishna  
Narayanan

Michel  
Schanen

Paul Hovland

TB Schardl

Praytush Das

Tim Gymnich

Albert Cohen

Sven  
Verdoolaege

Ruizhe Zhao



Manuel  
Drehwald

Nicolas  
Vasliache

Alex Zinenko

Theodoros  
Theodoridis

Priya Goyal

Ivan R. Ivanov

Jens Domke

Toshio Endo



Ameer  
Haj Ali

Jenny  
Huang

Ion  
Stoica

Krste  
Asanovic

John  
Wawrzynek

&

more



# Acknowledgements


---


- This work was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323. This research was supported in part by LANL grant 531711; in part by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357; in part by the Exascale Computing Project (17-SC-20-SC). Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.
- This work was funded and/or supported by NSF Cyberinfrastructure for Sustained Scientific Innovation (CSSI) award numbers: 2104068, 2103942, and 2103804, Argonne Leadership Computing Facility, which is a U.S. Department of Energy (DOE) Office of Science User Facility supported under Contract DE-AC02-06CH11357, NSF (grants OAC-1835443, AGS-1835860, and AGS-1835881), DARPA under agreement number HR0011-20-9-0016 (PaPPa), Schmidt Futures program, Paul G. Allen Family Foundation, Charles Trimble, Audi Environmental Foundation, DOE, National Nuclear Security Administration under Award Number DE-NA0003965, LANL grant 531711, and German Research Council (DFG) under grant agreement No. 326472365.
- 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.

# Conclusions

---

- Explosion of specialized software packages and hardware architectures -> scientists spending more time learning how to optimize programs and use platform-specific API's than working on their intended problem.
- Rather than burdening the user, compilers can automatically generate fast, portable, and composable code.

 Enzyme generates fast derivatives of programs needed for science and machine learning, *without user rewriting*

 Tapir understands the parallelism within programs, enabling existing optimizations to apply with minimal modification.

- All these tools are open source and used in academia and industry and in disciplines that range from climate science to physics to material science

**Questions?**



# 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;  
}
```

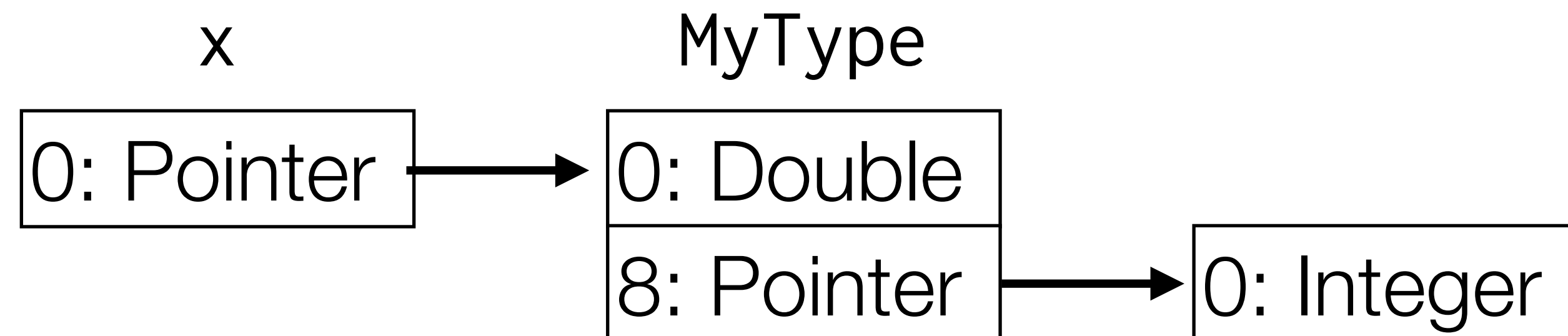
```
void grad_f(float* dst, float* dst',  
            float* src, float* src') {  
    // Forward Pass  
    memcpy(dst, src, 8);  
  
    // Reverse Pass  
    src'[0] += dst'[0];  
    dst'[0] = 0;  
    src'[1] += dst'[1];  
    dst'[1] = 0;  
}
```

# Type Analysis

---

- 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 MyType {  
    double;  
    int*;  
}  
  
x = MyType*;
```



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

# 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;  
}
```

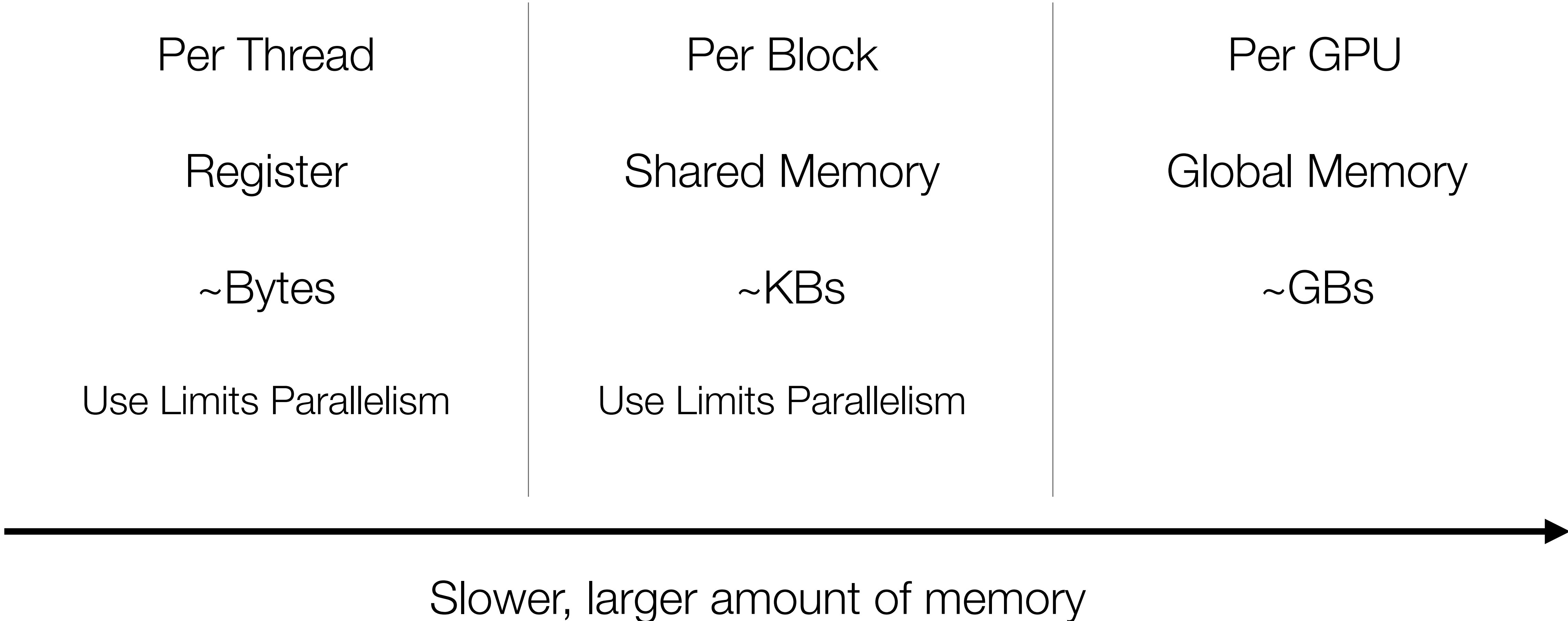
Read Race

```
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



# Synchronization Primitives

---

- Synchronization (`sync_threads`) ensures all threads finish executing `codeA` before executing `codeB`
- Sync is only necessary if A and B may access to the same memory
- Assuming the original program is race-free, performing a sync at the corresponding location in the reverse ensures correctness
- Prove correctness of algorithm by cases

```
codeA();  
sync_threads;  
codeB();
```

# Case 1: Store, Sync, Load

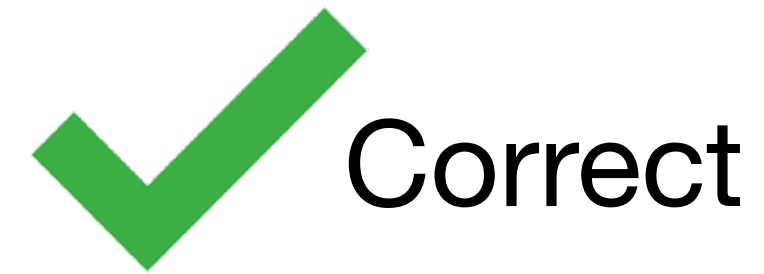
---

```
codeA(); // store %ptr
sync_threads;

codeB(); // load %ptr
...

diffe_codeB(); // atomicAdd %d_ptr
sync_threads;

diffe_codeA(); // load %d_ptr
                // store %d_ptr = 0
```



- Load of `d_ptr` must happen after all `atomicAdds` have completed

# CUDA Example

```
__device__
void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__device__
void __enzyme_autodiff(void*, ...);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {
    __enzyme_autodiff((void*)inner,
                     a, da, x, dx, y, dy);
}
```

```
__device__
void diffe_inner(float* a, float* da,
                float* x, float* dx,
                float* y, float* dy) {
    // Forward Pass

    y[threadIdx.x] = a[0] * x[threadIdx.x];

    // Reverse Pass

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    atomic { dx[threadIdx.x] += dx_tmp; }

    float da_tmp = x[threadIdx.x] * dy;
    atomic { da[0] += da_tmp; }
}
```

# CUDA Example

```
__device__
void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__device__
void __enzyme_autodiff(void*, ...);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {
    __enzyme_autodiff((void*)inner,
                     a, da, x, dx, y, dy);
}
```

```
__device__
void diffe_inner(float* a, float* da,
                 float* x, float* dx,
                 float* y, float* dy) {
    // Forward Pass

    y[threadIdx.x] = a[0] * x[threadIdx.x];

    // Reverse Pass

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    dx[threadIdx.x] += dx_tmp;

    float da_tmp = x[threadIdx.x] * dy;
    reduce_accumulate(&da[0], da_tmp);
}
```



# CUDA.jl / AMDGPU.jl Example

---

```
function compute!(inp, out)
    s_D = @cuStaticSharedMem eltype(inp) (10, 10)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@cuda grad_compute!(Duplicated(inp, d_inp),
                   Duplicated(out, d_out))
```

```
function compute!(inp, out)
    s_D = AMDGPU.alloc_special(...)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@rocm grad_compute!(Duplicated(inp, d_inp),
                   Duplicated(out, d_out))
```

See Below For Full Code Examples

<https://github.com/wsmoses/Enzyme-GPU-Tests/blob/main/DG/>

# Efficient GPU Code

---

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

```
// Forward Pass
out[i] = x[i] * x[i];
x[i] = 0.0f;

// Reverse (gradient) Pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

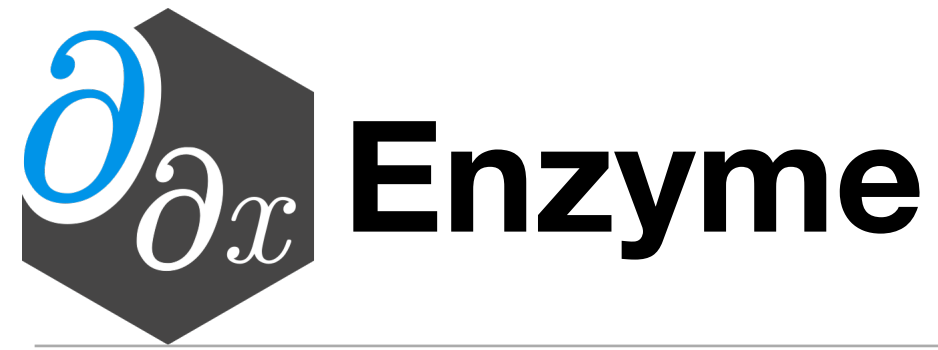
# Allocation Merging

---

- Allocations (and any calls) on the GPU are expensive
- Given two allocations in the same scope, replace uses with a single allocation
- Beneficial for not just AD, but any GPU programs!

```
double* var1 = new double[N];  
double* var2 = new double[M];  
  
use(var1, var2);  
  
delete[] var1;  
delete[] var2;
```

```
double* var1 = new double[N + M];  
double* var2 = var1 + N;  
  
use(var1, var2);  
  
delete[] var1;
```



- Tool for performing forward and 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 on CPU
- State-of-the art performance with existing tools
- First general purpose reverse-mode GPU AD
- Novel GPU and AD-specific optimizations improve runtime by several orders of magnitude
- Open source ([enzyme.mit.edu](http://enzyme.mit.edu) & join our mailing list)!
- Ongoing work to support Mixed Mode, Batching, Checkpointing/Scheduling

# 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);
}
```



# Cache

---

- Adjoint instructions may require values from the forward pass
  - e.g.  $\nabla(x * y) \Rightarrow x \, dy + y \, dx$
- For all values needed in the reverse, allocate memory in the forward pass to store the value
- Values computed inside loops are stored in an array indexed by the loop induction variable
  - Array allocated statically if possible; otherwise dynamically realloc'd

# When LLVM Doesn't Cut It

---

- Enzyme relies on optimizations such as LICM and CSE to eliminate redundant loads, and thus redundant caches.
- Since we instead need to preserve values for the reverse pass, these optimizations may not apply

```
for(int i=0; i<N; i++) {  
    for(int j=0; j<M; j++) {  
        use(array[j]);  
    }  
}  
overwrite(array);
```

# When LLVM Doesn't Cut It

- Enzyme relies on optimizations such as LICM and CSE to eliminate redundant loads, and thus redundant caches.
- Since we instead need to preserve values for the reverse pass, these optimizations may not apply
- This requires far more caching than necessary

```
double* cache = new double[N*M];

for(int i=0; i<N; i++) {
    for(int j=0; j<M; j++) {
        cache[i*M+j] = array[j];
        use(array[j]);
    }
}

overwrite(array);
grad_overwrite(array);

for(int i=0; i<N; i++) {
    for(int j=M-1; i<M; i++) {
        grad_use(cache[i*M+j], d_array[j]);
    }
}
```

# When LLVM Doesn't Cut It

- Enzyme relies on optimizations such as LICM and CSE to eliminate redundant loads, and thus redundant caches.
- Since we instead need to preserve values for the reverse pass, these optimizations may not apply
- This requires far more caching than necessary
- By analyzing the read/write structure, we can hoist the cache.

```
double* cache = new double[M];
memcpy(cache, array, sizeof(double)*M);
for(int i=0; i<N; i++) {
    for(int j=0; j<M; j++) {

        use(array[j]);
    }
}

overwrite(array);
grad_overwrite(array);

for(int i=0; i<N; i++) {
    for(int j=M-1; i<M; i++) {
        grad_use(cache[j], d_array[j]);
    }
}
```

# Cache

---

- Adjoint instructions may require values from the forward pass
  - e.g.  $\nabla(x * y) \Rightarrow x \, dy + y \, dx$
- For all values needed in the reverse, allocate memory in the forward pass to store the value
- Values computed inside loops are stored in an array indexed by the loop induction variable
  - Array allocated statically if possible; otherwise dynamically realloc'd



# Case Study: Read Sum

```
double sum(double* x) {  
    double total = 0;  
  
    for(int i=0; i<10; i++)  
        total += read() * x[i];  
  
    return total;  
}
```

```
void diffe_sum(double* x, double* xp) {  
    return __enzyme_autodiff(sum, x, xp);  
}
```

```
define double @sum(double* %x)
```

```
entry br for.body
```

```
%i = phi [ 0, entry ], [ %i.next, for.body ]  
%total = phi [ 0.0, %entry ], [ %add, for.body ]  
%call = @read()  
%0 = load %x[%i]  
%mul = %0 * %call  
%add = %mul + %total  
%i.next = %i + 1  
%exitcond = %i.next == 10  
br %exitcond, for.cleanup, for.body
```

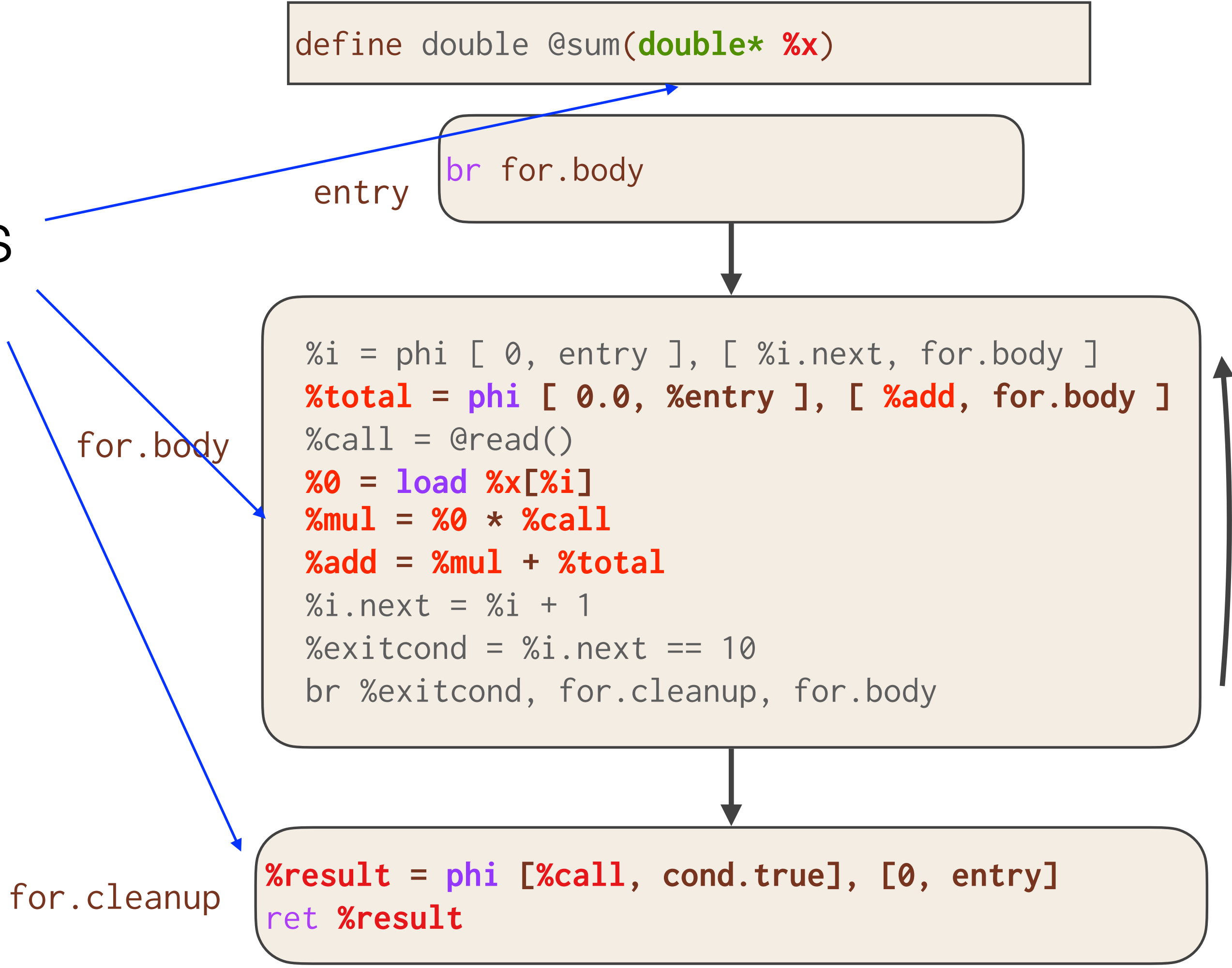
for.body

for.cleanup

```
%result = phi [ %call, cond.true ], [ 0, entry ]  
ret %result
```

# Case Study: Read Sum

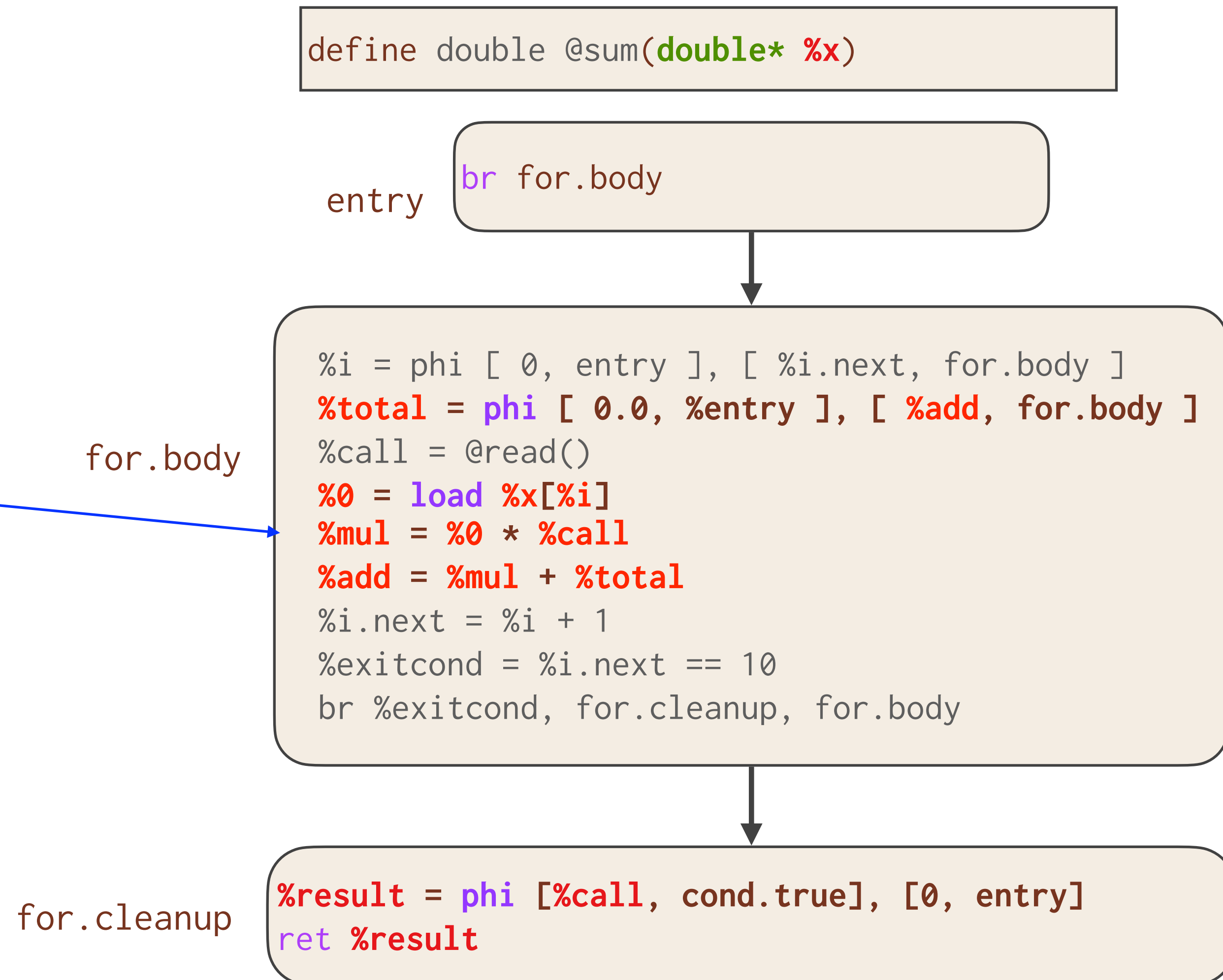
Active Variables



# Case Study: Read Sum

---

Each register in the for loop represents a distinct active variable every iteration



Allocate & zero  
shadow memory  
per active value

```
define double @diffe_sum(double* %x, double* %xp)
```

```
alloca %x'      = 0.0  
alloca %total'  = 0.0  
alloca %0'      = 0.0  
alloca %mul'    = 0.0  
alloca %add'    = 0.0  
alloca %result' = 0.0  
br for.body
```

entry

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]  
%total = phi [ 0.0, %entry ], [ %add, for.body ]  
%call = @read()  
%0 = load %x[%i]  
%mul = %0 * %call  
%add = %mul + %total  
%i.next = %i + 1  
%exitcond = %i.next == 10  
br %exitcond, for.cleanup, for.body
```

for.cleanup

```
%result = phi [ %call, cond.true ], [ 0, entry ]  
ret %result
```

```
define double @diffe_sum(double* %x, double* %xp)
```

entry

```
alloca %x'      = 0.0  
alloca %total'  = 0.0  
alloca %0'      = 0.0  
alloca %mul'    = 0.0  
alloca %add'    = 0.0  
alloca %result' = 0.0  
%call_cache = @malloc(10 x double)  
br for.body
```

```
%i = phi [ 0, entry ], [ %i.next, for.body ]  
%total = phi [ 0.0, %entry ], [ %add, for.body ]  
%call = @read()  
store %call_cache[%i] = %call  
%0 = load %x[%i]  
%mul = %0 * %call  
%add = %mul + %total  
%i.next = %i + 1  
%exitcond = %i.next == 10  
br %exitcond, for.cleanup, for.body
```

```
%result = phi [ %call, cond.true ], [ 0, entry ]  
@free(%cache)  
ret %result
```

for.body

for.cleanup

Cache forward pass  
variables for use in  
reverse



```
define void @diffe_sum(double* %x, double* %xp)
```

After lowering &  
some optimizations

entry

```
%call_cache = @malloc(10 x double)  
br for.body
```

for.body

```
%i = phi [ 0, entry ], [ %i.next, for.body ]  
%total = phi [ 0.0, %entry ], [ %add, for.body ]  
%call = @read()  
store %call_cache[%i] = %call  
%i.next = %i + 1  
%exitcond = %i.next == 10  
br %exitcond, reversefor.body, for.body
```

reversefor.body

```
%i' = phi [ 9, for.body ], [ %i'.next, reversefor.body ]  
%i'.next = %i' - 1  
%cached_read = load %call_cache[%i']  
store %xp[%i'] = %cached_read + %xp[%i']  
%exit2 = %i = 0  
br %exitcond, %exit2, reversefor.body
```

exit

```
@free(%cache)  
ret
```

# Case Study: Read Sum

```
define void @diffe_sum(double* %x, double* %xp)
```

entry

```
%call0 = @read()
store %xp[0] = %call0
%call1 = @read()
store %xp[1] = %call1
%call2 = @read()
store %xp[2] = %call2
%call3 = @read()
store %xp[3] = %call3
%call4 = @read()
store %xp[4] = %call4
%call5 = @read()
store %xp[5] = %call5
%call6 = @read()
store %xp[6] = %call6
%call7 = @read()
store %xp[7] = %call7
%call8 = @read()
store %xp[8] = %call8
%call9 = @read()
store %xp[9] = %call9
ret
```

After more  
optimizations

```
void diffe_sum(double* x, double* xp) {
    xp[0] = read();
    xp[1] = read();
    xp[2] = read();
    xp[3] = read();
    xp[4] = read();
    xp[5] = read();
    xp[6] = read();
    xp[7] = read();
    xp[8] = read();
    xp[9] = read();
}
```

# CUDA Automatic Differentiation

---

- Enzyme enables differentiation of CPU programs without rewriting them in a DSL.
- Similarly, GPU programs cannot currently be differentiated without being rewritten in a differentiable language (e.g. PyTorch).
- Enzyme enables reverse-mode AD of general existing GPU programs by:
  - Resolving potential data race issues
  - Differentiating parallel control (syncthreads)
  - Differentiating CUDA intrinsics (e.g. `threadIdx.x /llvm.nvvm.read.ptx.sreg.tid.x`)
  - Handling shared memory

# CUDA Automatic Differentiation

---

- Most CUDA intrinsics [e.g. threadIdx.x] are inactive and recomputable and thus are incorporated into Enzyme without any special handling
- Derivative of syncthreads is a syncthreads at the corresponding place in reverse pass
- Shared memory is handled by making a second shared memory allocation to act as the shadow for any potentially active uses

# 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



# Activity Analysis

---

- Determines what instructions could impact derivative computation
- Avoids taking meaningless or unnecessary derivatives (e.g.  $d/dx$  cpuid)
- Instruction is active iff it can propagate a differential value to its return or memory
- Build off of alias analysis & type analysis
  - E.g. all read-only function that returns an integer are inactive since they cannot propagate adjoints through the return or to any memory location

# Compiler Analyses Better Optimize AD

---

- Existing
- Alias analysis results that prove a function does not write to memory, we can prove that additional function calls do not need to be differentiated since they cannot impact the output
- Don't cache equivalent values
- Statically allocate caches when a loop's bounds can be determined in advance

# Decomposing the “Tape”

---

- Performing AD on a function requires data structures to compute
  - All values necessary to compute adjoints are available [cache]
  - Place to store adjoints [shadow memory]
  - Record instructions [we are static]
- Creating these directly in LLVM allows us to explicitly specify their behavior for optimization, unlike approaches that call out to a library
- For more details look in paper

# Conventional Wisdom: AD Only Feasible at High-Level

---

- Automatic Differentiation requires high level semantics to produce gradients
- Lack of high-level information can hinder performance of low-level AD
  - “AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR [...]” -Innes<sup>[1]</sup>

[1] Michael Innes. Don't Unroll Adjoint: Differentiating SSA-Form Programs. arXiv preprint arXiv:1810.07951, 2018

# Differentiation Is Key To Machine Learning

```
// C++ nbody simulator

void step(std::array<Planet> bodies, double dt) {
    vec3 acc[bodies.size()];
    for (size_t i=0; i<bodies.size(); i++) {
        acc[i] = vec3(0, 0, 0);
        for (size_t j=0; j<bodies.size(); j++) {
            if (i == j) continue;
            acc[i] += force(bodies[i], bodies[j]) /
                    bodies[i].mass;
        }
    }
    for (size_t i=0; i<bodies.size(); i++) {
        bodies[i].vel += acc[i] * dt;
        bodies[i].pos += bodies[i].vel * dt;
    }
}
```

```
// PyTorch rewrite of nbody simulator
import torch

def step(bodies, dt):
    acc = []
    for i in range(len(bodies)):
        acc.push(torch.zeros([3]))
        for j in range(len(bodies)):
            if i == j: continue
            acc[i] += force(bodies[i], bodies[j]) /
                    bodies[i].mass

    for i, body in enumerate(bodies):
        body.vel += acc[i] * dt
        body.pos += body.vel * dt
```

- Hinders application of ML to new domains
- Synthesizing gradients aims to close this gap



## Case 3: Store, Sync, Store

---

```
codeA(); // store %ptr
sync_threads;

codeB(); // store %ptr
...
diffe_codeB(); // load %d_ptr
                // store %d_ptr = 0

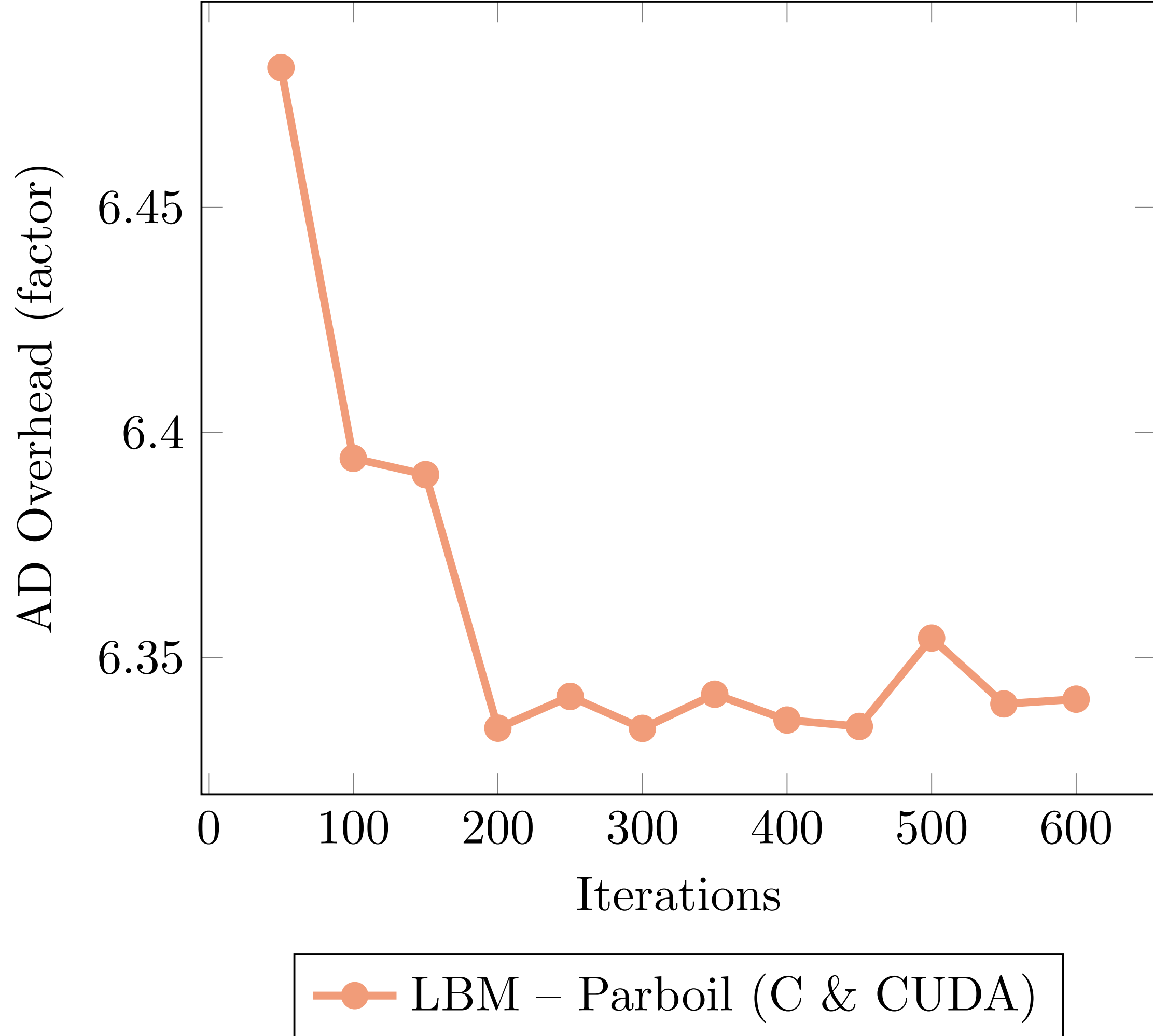
sync_threads;

diffe_codeA(); // load %d_ptr
                // store %d_ptr = 0
```



- All stores to `d_ptr` in `diffe_B` will complete prior to `diffe_A`, ensuring only the clobbering store has its derivative incremented

# Scalability Analysis (Fixed Thread Count)



# CUDA Example

---

```
__device__ void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}
__device__ void __enzyme_autodiff(void*, ...);

__global__ void daxpy(float* a, float* da, float* x, float* dx, float* y, float* dy) {
    __enzyme_autodiff((void*)inner, a, da, x, dx, y, dy);
}
```

```
__device__ void diffe_inner(float* a, float* da, float* x, float* dx, float* y, float* dy) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    atomic { dx[threadIdx.x] += dx_tmp; }

    float da_tmp = x[threadIdx.x] * dy;
    atomic { da[0] += da_tmp; }
}
```

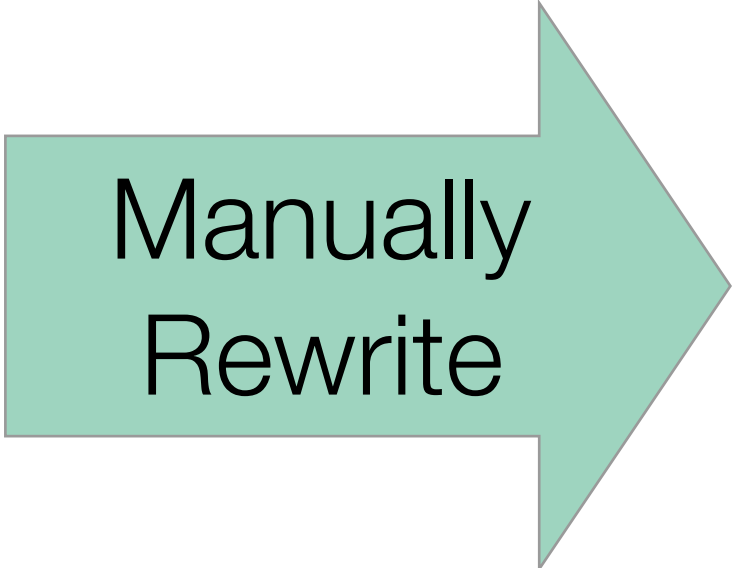
# Existing AD Approaches (1/3)

---

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

Manually  
Rewrite



```
import tensorflow as tf  
  
x = tf.Variable(3.14)  
  
with tf.GradientTape() as tape:  
    out = tf.math.square(x)  
  
print(tape.gradient(out, x).numpy())
```

# 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

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

Tool  
Rewrite



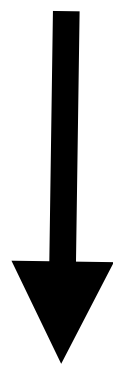
```
double grad_square(double val) {  
    return 2 * val;  
}
```

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



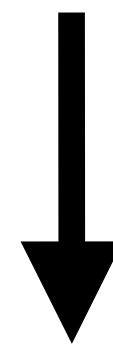
# Parallel Automatic Differentiation in LLVM

```
%res = load %ptr
```



```
%tmp = load %d_res  
store %d_res = 0  
atomic %d_ptr += %tmp
```

```
store %ptr = %val
```



```
%tmp = load %d_ptr  
store %d_ptr = 0  
load/store %d_val += %tmp
```

- Shadow Registers `%d_res` and `%d_val` are ***thread-local*** as they shadow thread-local registers.
- No risk of races and no special handling required.
- Both `%ptr` and shadow `%d_ptr` might be raced upon and require analysis.

## Case 2: Load, Sync, Store

---

```
codeA(); // load %ptr
sync_threads;

codeB(); // store %ptr
...

diffe_codeB(); // load %d_ptr
                // store %d_ptr = 0

sync_threads;

diffe_codeA(); // atomicAdd %d_ptr
```



Correct

- All of the stores of `d_ptr` will complete prior to any `atomicAdds`

No cross-thread race here since that's equivalent to a write race in B

# Differentiation of SyncThreads

## Case 3 [write sync write]

```
codeA(); // store %ptr
sync_threads;
codeB(); // store %ptr
...
diffe_codeB(); // load %d_ptr
                // store %d_ptr = 0
sync_threads;
diffe_codeA(); // load %d_ptr
                // store %d_ptr = 0
```

All uses of stores to `d_ptr` in `diffe_B` will correctly complete prior to `diffe_A`



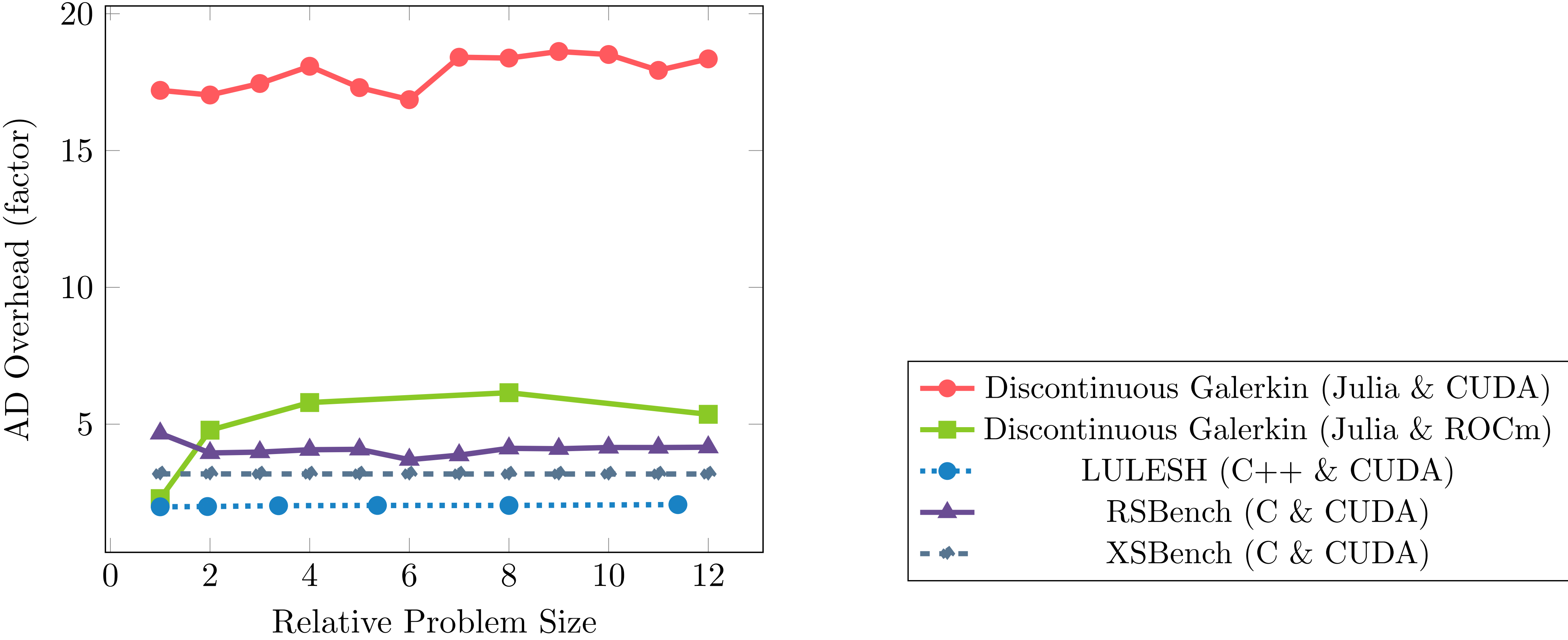
## Case 4 [read sync read]

```
codeA(); // load %ptr
sync_threads;
codeB(); // load %ptr
...
diffe_codeB(); // atomicAdd %d_ptr
sync_threads;
diffe_codeA(); // atomicAdd %d_ptr
```

Original and differential sync unnecessary and legal to include



# Scalability Analysis (Fixed Work Per Thread)



# Efficient Gradient Code

---

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - Complex memory hierarchies, like on the GPU, cause caches to slow down the program by several orders of magnitude, if they even fit at all
- Existing optimizations reduce the overhead, but may not be sufficient
- Novel AD-specific optimizations can speedup by several orders of magnitude

```
// Forward Pass
out[i] = x[i] * x[i];
x[i] = 0.0f;

// Reverse (gradient) Pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

# Efficient Correct Gradient Code

- For correctness, Enzyme may need to cache values in order to compute the gradient
  - Complex memory hierarchies, like on the GPU, cause caches to slow down the program by several orders of magnitude, if they even fit at all
- Existing optimizations reduce the overhead, but may not be sufficient
- Novel AD-specific optimizations can speedup by several orders of magnitude

```
double* x_cache = new double[...];

// Forward Pass

out[i] = x[i] * x[i];
x_cache[i] = x[i];

x[i] = 0.0f;

// Reverse (gradient) Pass

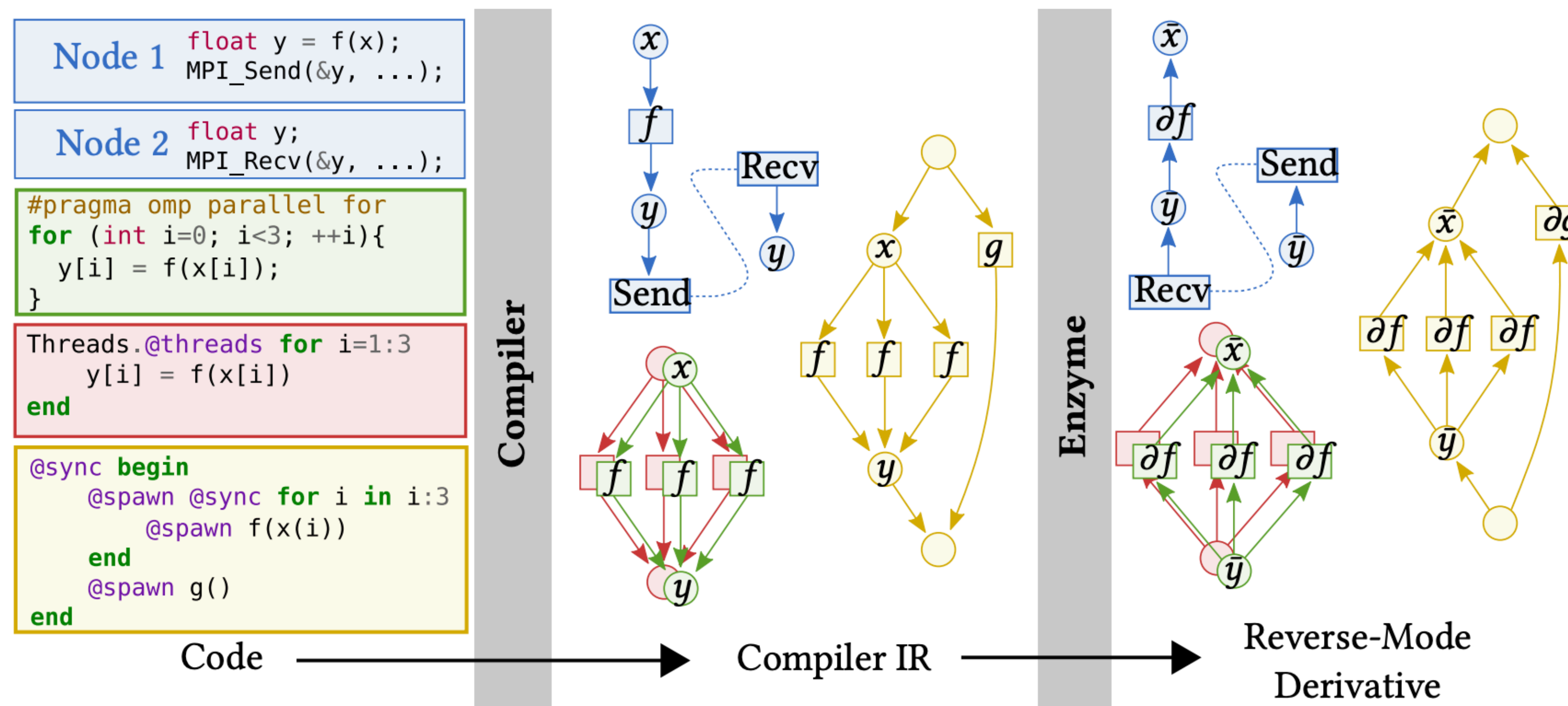
...
grad_x[i] += 2 * x_cache[i]
             * grad_out[i];
...

delete[] x_cache;
```



# Common Framework for Parallel AD (SC'22, Best Student Paper)

- Common infrastructure for supporting parallel AD (caching, race-resolution, gradient accumulation) enables parallel differentiation independent of framework or language.



- Enables differentiation of a combination of GPU (e.g. CUDA, ROCm), CPU (OpenMP, Julia Tasks, RAJA), Distributed (MPI, MPI.jl), and more



# History of Parallel AD

- Prior AD tools are built with a single language and parallel framework in mind
  - Differentiating code using multiple parallel frameworks is difficult or impossible!
- Require AD-specific rewriting to specify extra information
- Run at a source-level, preventing optimizations from being applied



MPI AD



```
void send(double* data, int size) {  
    MPI_Isend(data, val);  
}
```



OpenMP AD



```
void send(ADdouble* data, int size, void* buffer) {  
    AD_MPI_Isend(data, val, buffer);  
}
```



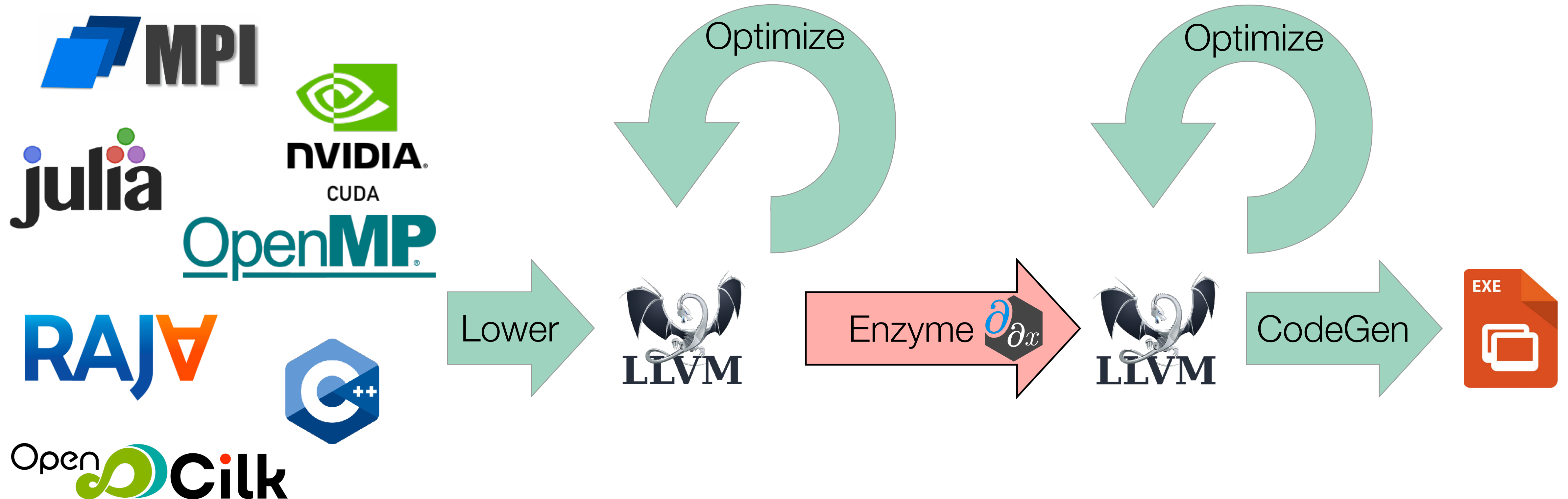
CUDA AD





# Combining Parallelism with Differentiation

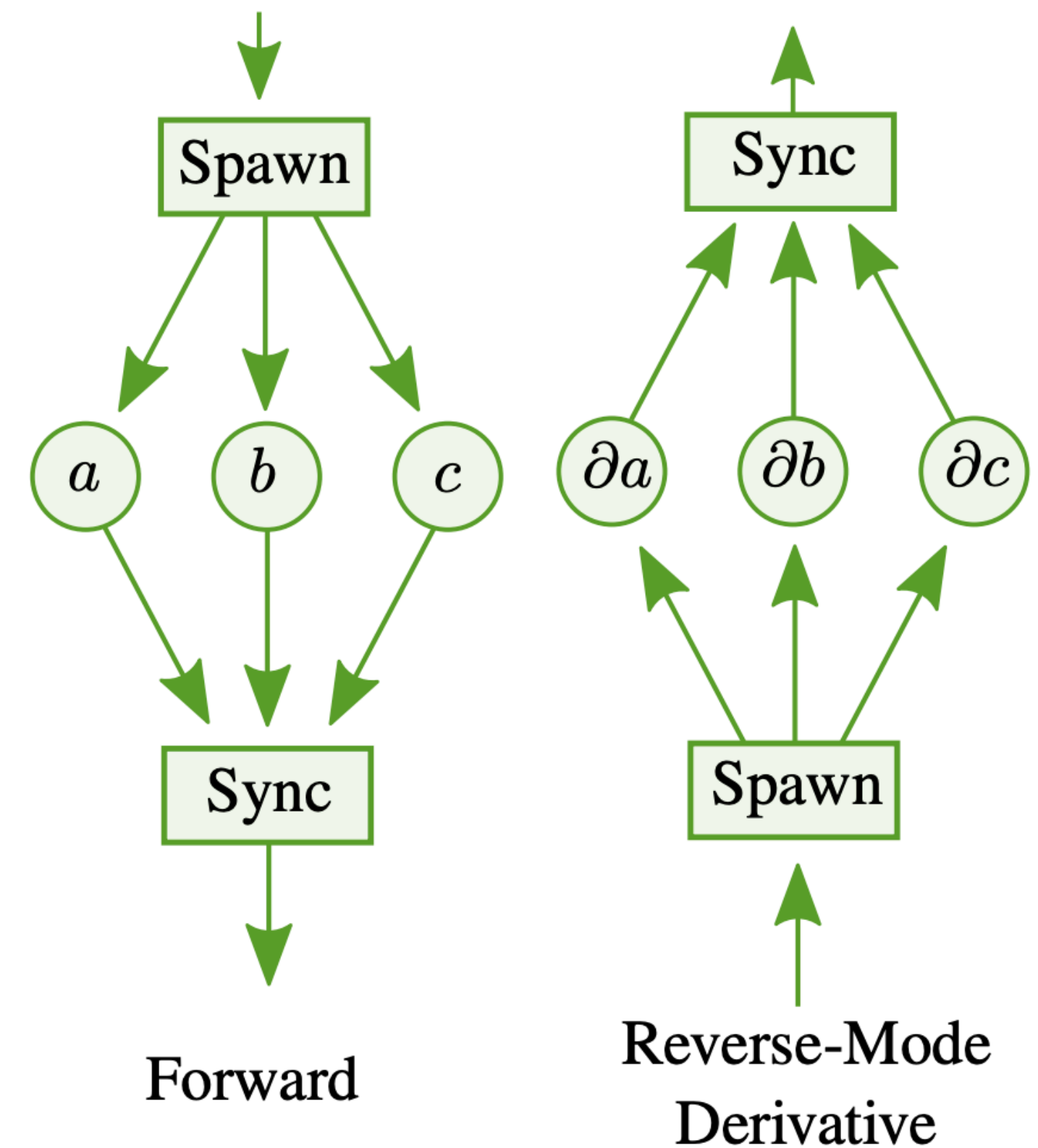
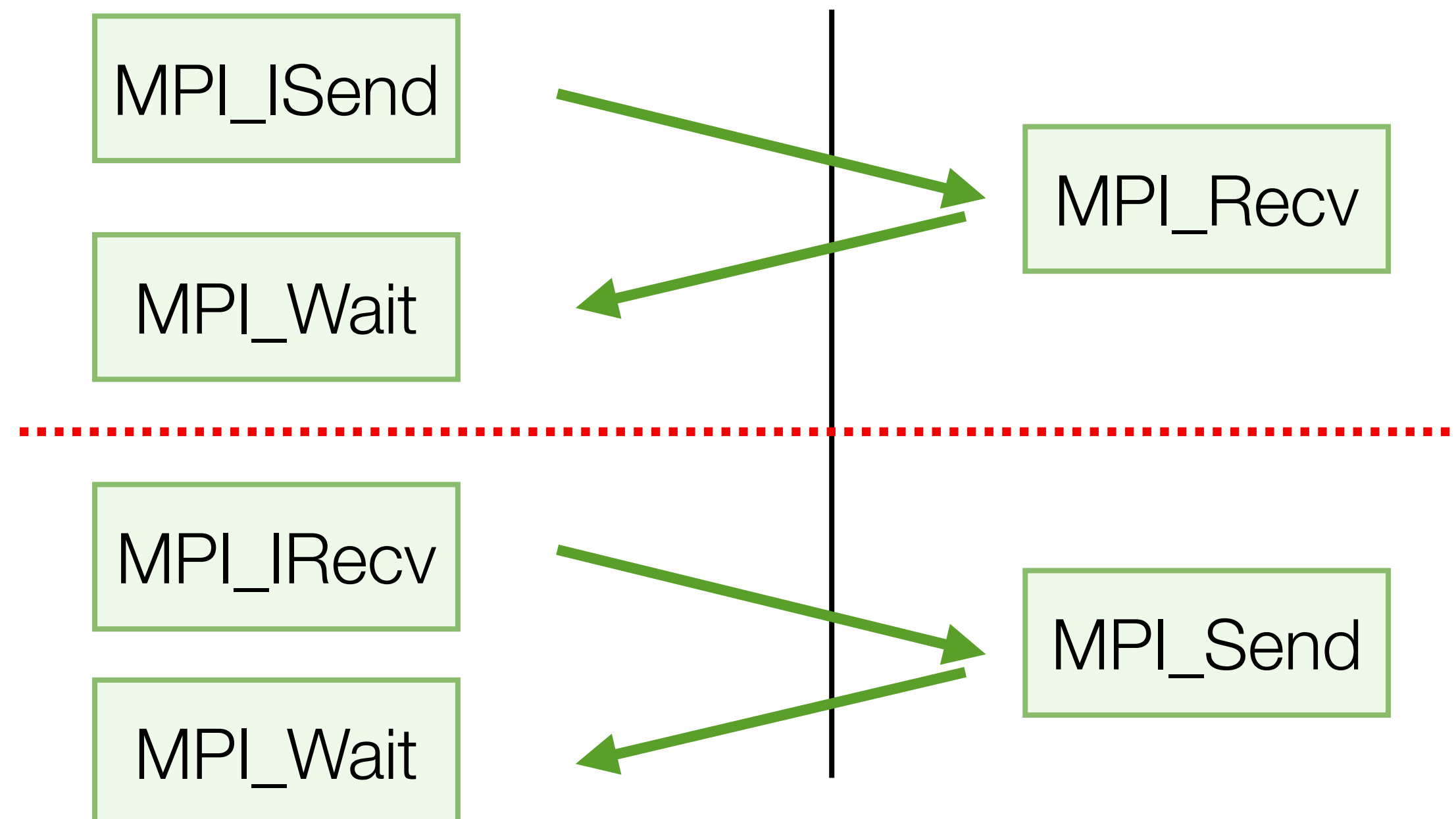
Performing AD in the compiler lets us build a common tool to differentiate & optimize multiple parallel frameworks simultaneously!





# Parallelism-Preserving Differentiation

- Computing the adjoint of an instruction in the reverse pass updates the derivative of the operands it used.
- Reversing the parallel dependency structure ensures that for a given value all derivative updates are performed before its definition



# Data Caching

- Differentiation requires some values from the original program for correctness
- Overwriting a value required for the derivative requires it to be cached
- Recomputing a value can significantly reduce both memory overhead and runtimes, if legal
- Parallel constructs (closures, thread-local vs global memory) hinder such optimizations
- Remedy via novel parallel analyses and optimizations

```
// Forward Pass
out[i] = x[i] * x[i];
x[i] = 0.0f;

// Reverse (gradient) Pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```





# Parallel Value Hoisting

```
#pragma omp parallel for
for(int i=0; i<10; i++) {
    out[i] = in[i] * in[i];
}
```

```
void closure(double** outp, double** inp) {
    // Unknown aliasing between out/in
    double* out = *outp;
    double* in = *inp;
    int i = threadid();
    out[i] = in[i] * in[i];
}
```

...

```
double** outp = &out;
double** inp = &in;
```

```
kmpc_fork(closure, outp, inp);
```

```
void closure(double* restrict out2,
             double* restrict in2) {

    // out/in known to not overlap
    out2[i] = in2[i] * in2[i];
}
```

...

```
double** outp = &out;
double** inp = &in;
double* out2 = *outp;
double* in2 = *inp;
kmpc_fork(closure, out2, inp2);
```





# Parallel Value Hoisting

```
#pragma omp parallel for
for(int i=0; i<10; i++) {
    out[i] = in[i] * in[i];
}
```

```
void closure(double** outp, double** inp) {
    // Unknown aliasing between out/in
    double* out = *outp;
    double* in = *inp;
    int i = threadid();
    out[i] = in[i] * in[i];
}
```

...

```
double** outp = &out;
double** inp = &in;
```

```
kmpc_fork(closure, outp, inp);
```

```
void closure(double* restrict out2,
             double* restrict in2) {

    // out/in known to not overlap
    out2[i] = in2[i] * in2[i];
}
```

...

```
double** outp = &out;
double** inp = &in;
double* out2 = *outp;
double* in2 = *inp;
kmpc_fork(closure, out2, in2);
```

# Framework Generality

---

- Implemented hooks for several parallel frameworks:
  - OpenMP
  - MPI
  - Julia Tasks
  - existing GPU support (ROCM, CUDA)
- Supports any higher-level framework built off these primitives
  - RAJA
  - MPI.jl
  - Julia @parallel
  - ...



# Construct Generality

---

- Higher-level parallel utilities are automatically handled by existing support for parallelism
  - Both source-level or manually written utilities are lowered to common form.
- If optimizations exist for higher-level utilities, Enzyme supports overriding
  - E.g. faster OpenMP *parallel for*, rather than differentiating via separate support for OpenMP *parallel* and work sharing loop

```
double min_per_thread[num_threads()];
#pragma omp parallel
{
    double min_value = 0;
    #pragma omp for
    for(int i = 0; i < N; i++)
        min_value = min(data[i], min_value);
    min_per_thread[omp_get_thread_num()] = min_value;
}
double final_val = 0;
for(int i = 1; i < omp_get_num_threads(); i++)
    final_val = min(final_val, min_per_thread[i]);
```



# Evaluation

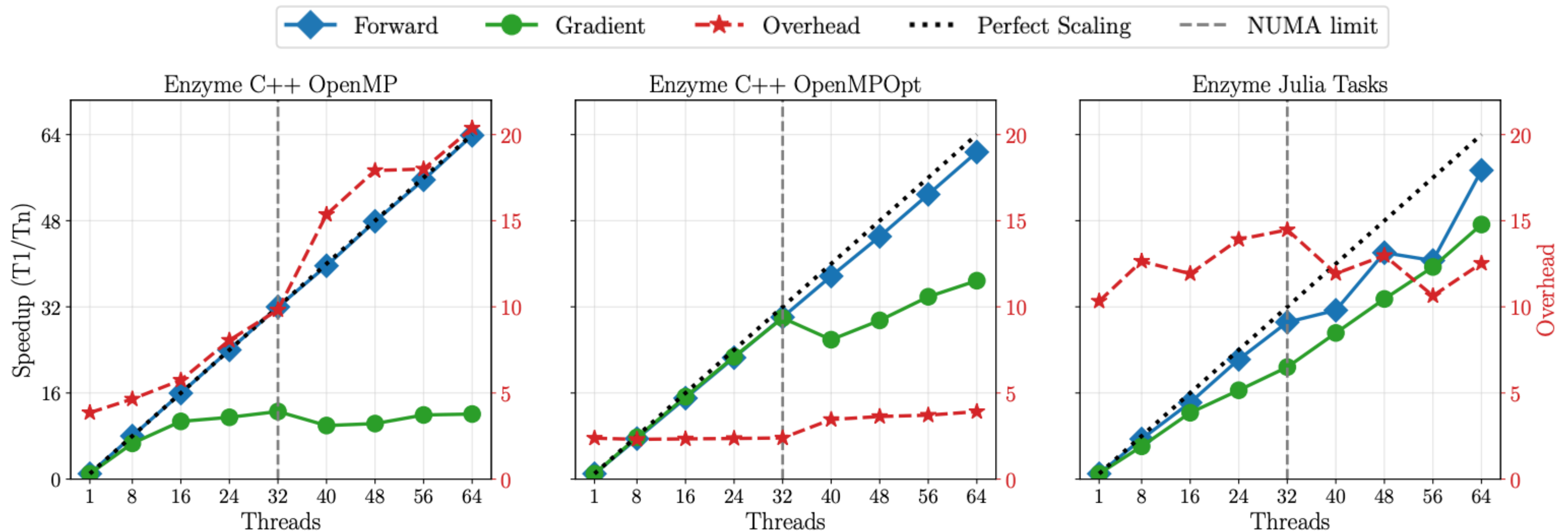
---

- Differentiated nine distinct versions of LULESH and miniBUDE applications, in a variety of parallel frameworks, and in both C++ and Julia
  - LULESH: unstructured hydrodynamics solver
  - miniBUDE: computational kernels of a molecular docking engine
- Compare performance and scalability against non-differentiated code, as well as a state of the art MPI AD tool (CoDiPack)
- Benchmarks available at: <https://github.com/EnzymeAD/Enzyme-sc22>



# Evaluation Highlights: Strong Scaling (BUDE)

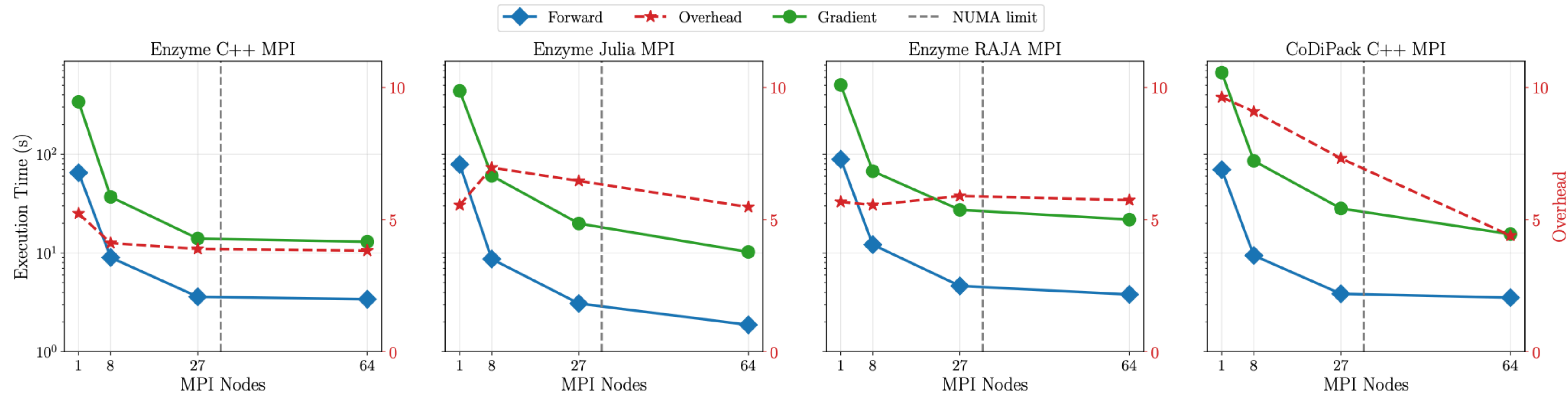
- Parallel optimizations enable Enzyme to keep the same scalability as the original program





# Evaluation Highlights: Runtime Overhead (LULESH)

- Overhead is stable and small, independent of number of MPI nodes, or language/framework







- Tool for performing reverse-mode (and forward mode) AD of statically analyzable LLVM IR
- Differentiates code in a variety of parallel frameworks (OpenMP, MPI, Julia Tasks, GPU), and languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- Parallel and AD-specific optimizations crucial for performance
- Keep similar scalability as non-differentiated code
- Open source ([enzyme.mit.edu](http://enzyme.mit.edu) & join our mailing list)!
- Ongoing work to support Mixed Mode, Batching, Checkpointing, and more



# Enzyme: Fast, Parallel, and Rewrite-Free Derivatives

- Derivatives are ubiquitous in machine learning (training neural networks, Bayesian inference), scientific computing (uncertainty quantification, simulation)
- Enzyme synthesizes derivatives of arbitrary code within the compiler
  - Differentiate code in any LLVM-based language (C/C++, Julia, Rust, Swift, Fortran, Python, etc) *without rewriting it!*
  - Operating after and alongside program optimization generates asymptotically and empirically faster derivatives
  - First automatic differentiation tool to handle arbitrary GPU kernels
- **Best student paper @SC'22, SC'21, spotlight @NeurIPS'20; awarded multi-year DOE grant with LLNL**
- Used by Harvard, Facebook, AMD, ANL, UT Austin, NASA, Dartmouth, CU Boulder, TU Munich, and startups for climate simulation, material science, ML, and more!

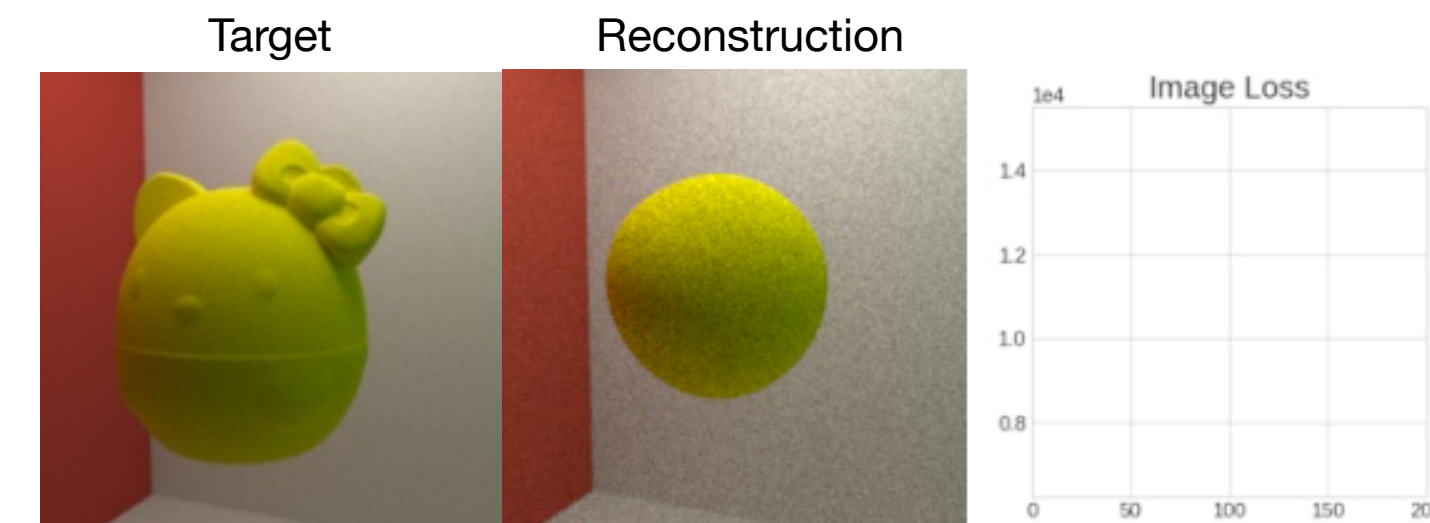


>100x speedup!

Prior:  
**5 days (cluster)**

Enzyme-Based:  
**1 hour (laptop)**

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



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

# Teaching: Combining Theory with Practice

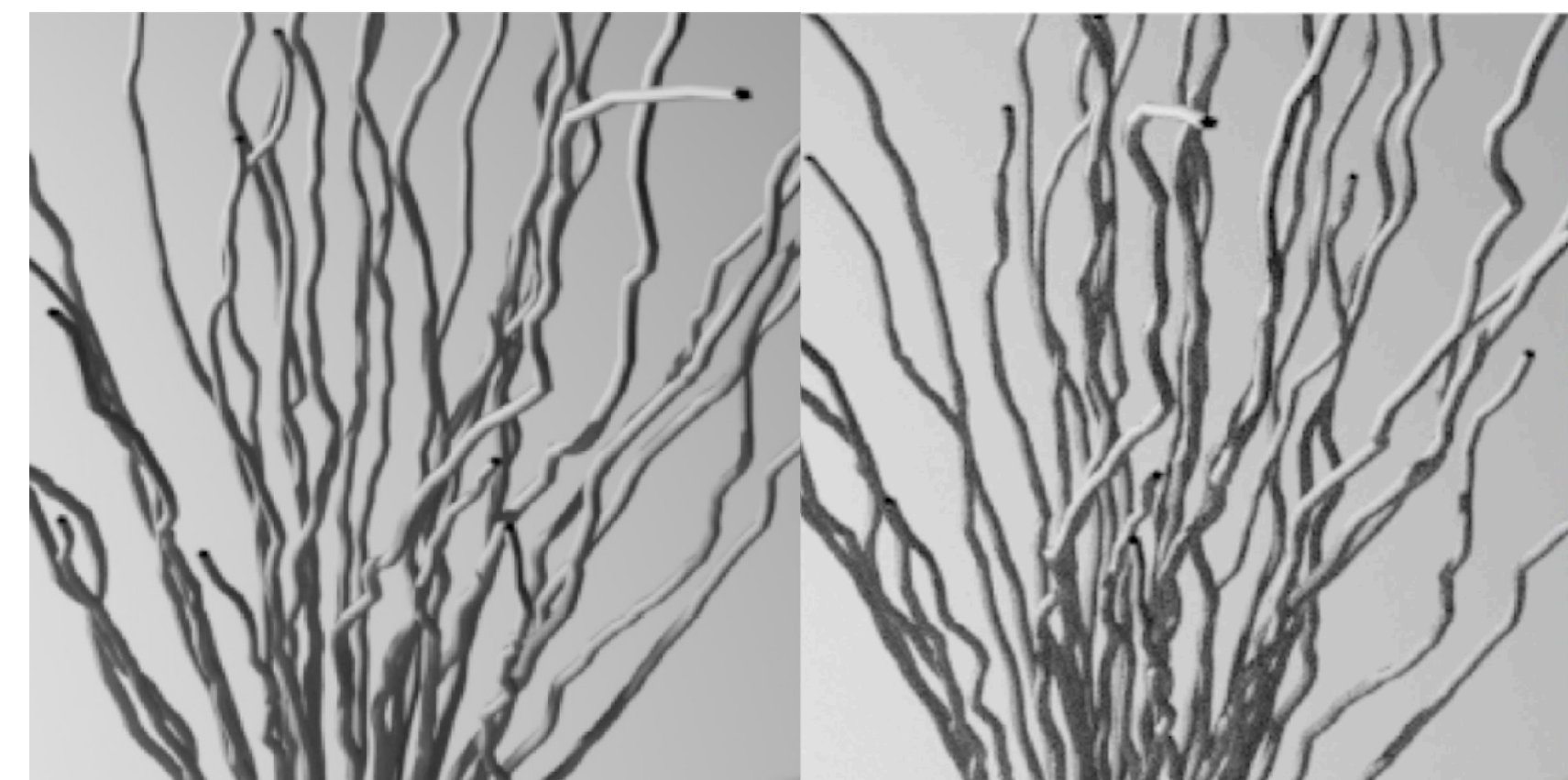
---

*My goal is to teach my students the principles behind modern systems and provide them with a foundation for understanding any future systems they may encounter or even build themselves.*

- Prior Experience: MIT Intro to Algorithms (twice-weekly recitations); created January mini-term C/C++ course; guest lecture for graduate Data Analysis & Signal Processing course; and more

## New Courses:

- *Differential Programming*: Code transformations enable using code as a component of ML models. Gradient descend through a physics simulation to find an optimal aircraft wing design! The course will both teach foundational algorithms and provide experience writing real differentiable programs.
- *Parallel Performance Engineering*: Modern computing requires efficiently using the performance of multicore chips, clusters, and accelerators. Learn about both hardware constraints like pipeline and caches and software constraints like allocators, needed to build and debug fast code.



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



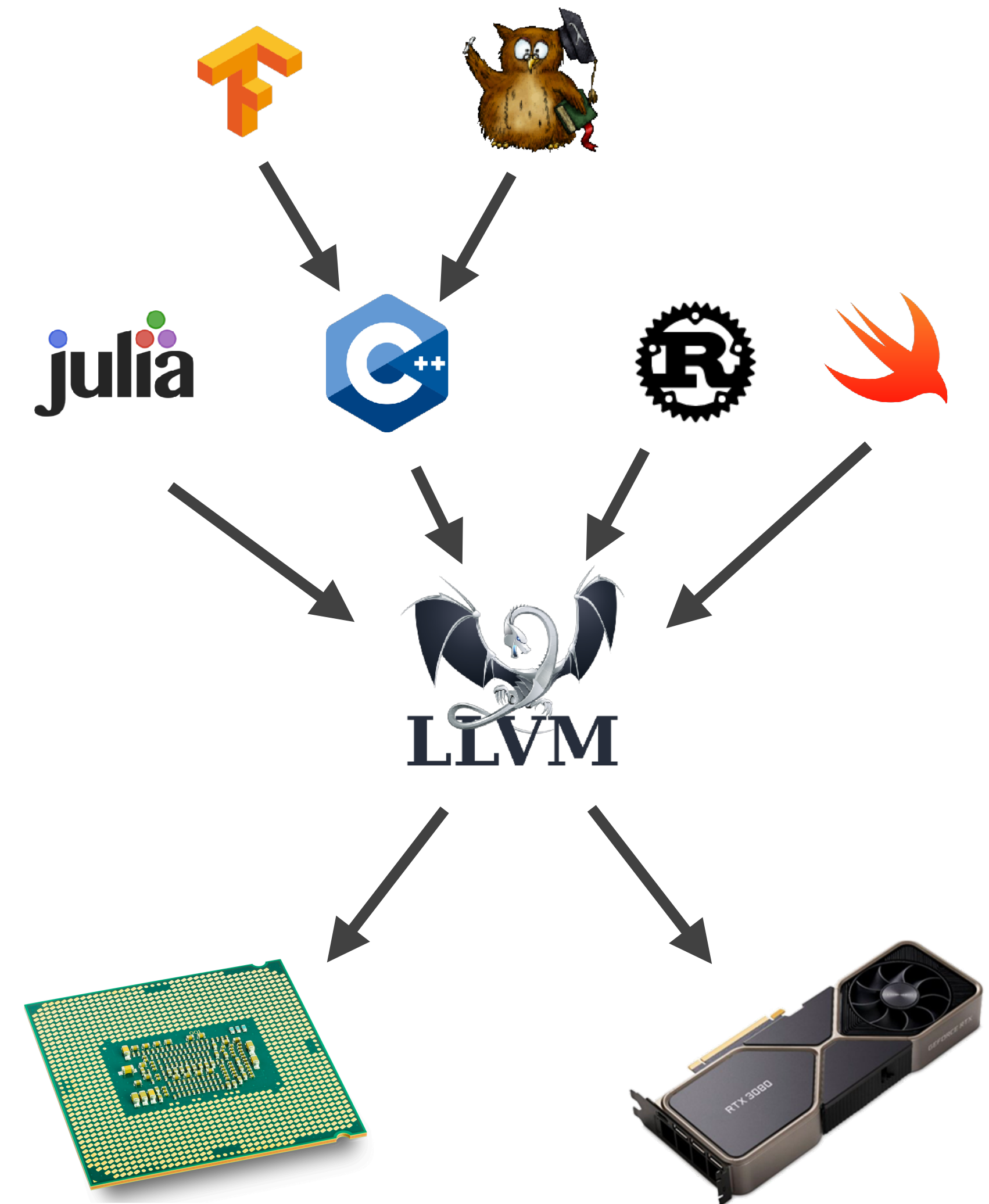
# Diversity Equity and Inclusion

---

- It is the duty of ***all professors*** to promote diversity, equity, and inclusion within Princeton and the broader community.
  - Faculty members of systemically marginalized groups are historically expected to perform most inclusion work, and are simultaneously judged more harshly spending less time on research as a result.
- Prior Experience:
  - Worked in MIT's Institute Community and Equity Office to support DEI initiatives. For example, I created opportunities for students from disadvantaged backgrounds to interact with tenured faculty, arranged speakers, wrote news articles.
  - As president of MIT's oldest computing club, I improved diversity and general attendance by 20% through various initiatives (mentorship program, outreach, culture of positive learning, community-building projects). Awarded the Golden Beaver and ***Karl Taylor Compton Prize, MIT's highest student award***.
- Future Initiatives:
  - Studies have shown that mentorship programs are some of the most effective methods for improving diversity.
  - Propose long-term research mentorship program for local high-school students, and cohort and other community building for undergraduate and graduate students; low-barrier anonymous feedback in courses, research group, and department

# Why Does Enzyme Use LLVM?

- Generic low-level compiler infrastructure with many frontends
  - “Cross platform assembly”
  - Many backends (CPU, CUDA, AMDGPU, etc)
- Well-defined semantics
- Large collection of optimizations and analyses





# Implementing Tapir/LLVM

| <i>Compiler component</i> | <i>LLVM 4.0svn (lines)</i> | <i>Tapir/LLVM (lines)</i> |         |
|---------------------------|----------------------------|---------------------------|---------|
| Instructions              | 105,995                    | 943                       | } 1,768 |
| Memory behavior           | 21,788                     | 445                       |         |
| Optimizations             | 152,229                    | 380                       |         |
| Parallelism lowering      | 0                          | 3,782                     |         |
| Other                     | 3,803,831                  | 460                       |         |
| <b>Total</b>              | <b>4,083,843</b>           | <b>6,010</b>              |         |



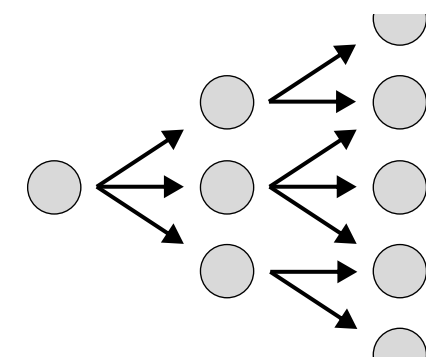
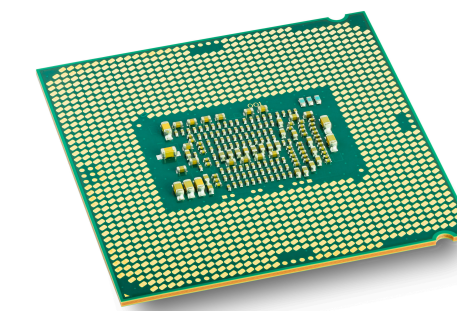
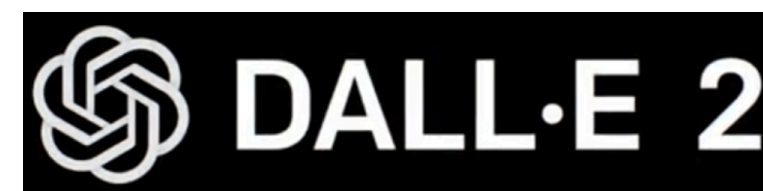
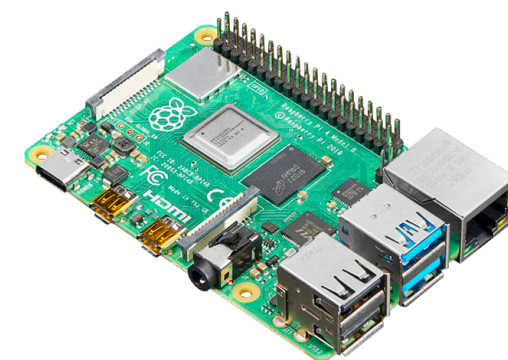
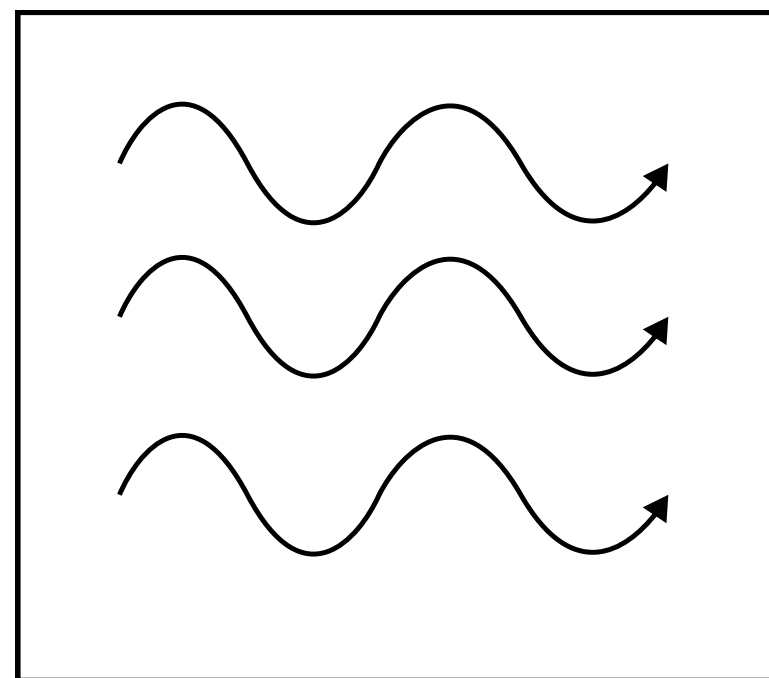
# Revisiting The Programmer's Burden



$$A\vec{x}$$



L A P A C K  
L -A P -A C -K  
L A P A -C -K  
L -A P -A -C K  
L A -P -A C K  
L -A -P A C -K



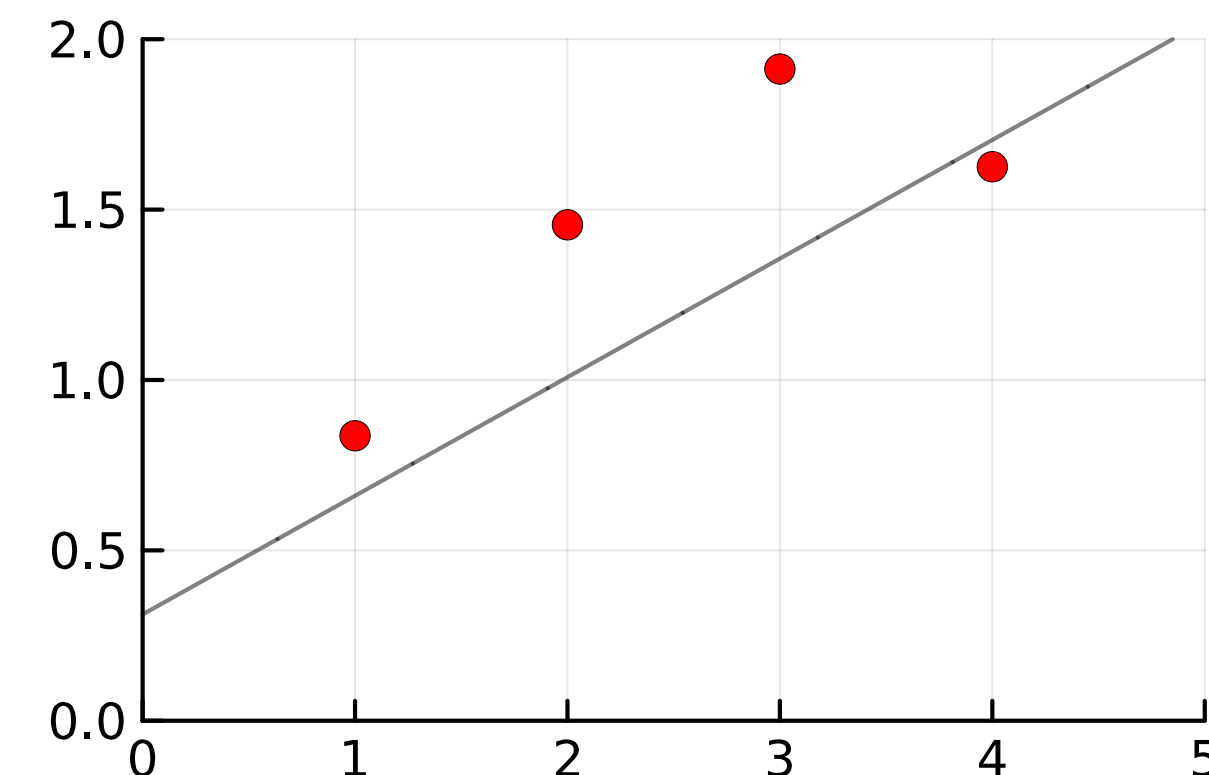
# Key Enabling Technology: Probabilistic Programming

*Probabilistic Programming* is a new paradigm for automating statistical and Bayesian reasoning

- Use requires rewriting entire applications in a probabilistic programming language (PPL), with analyses performed on source code, if at all
- Inference requires running model functions many times, even if variables won't change the results.
- Idea: Moving into the compiler will enable performance and usability advantages.

```
@gen function model(N)
  m = @trace(normal(0.0, 1.0), :m)
  b = @trace(normal(0.0, 1.0), :b)
  predictions = []
  for i in 1:N
    push!(predictions,
          @trace(normal(i * m + b, 1.0), (:predict, i)))
  end
  return m, b, predictions
end

plot(simulate(model, 4))
```



# Generalizing Support: Library-Specific Optimization

---

- All libraries have high-level semantics or properties that are not well-expressed within a given programming language -> failure to optimize
- Provide lightweight source-level mechanisms that enable library-authors preserve, optimize, and verify custom semantics

```
void foo(DataStructure& x) {  
    print(size(x));  
    insert(x);  
    print(size(x));  
}
```

```
define void @foo(ptr %x) {  
    %2 = call @size(ptr %x)  
    call @print(i32 %2)  
    call @insert(ptr %x)  
    ; %3 = add i32 %2, 1  
    %3 = call @size(ptr %x)  
    call @print(i32 %3)  
    ret void  
}
```