Supercharging Programming Through Compiler Technology

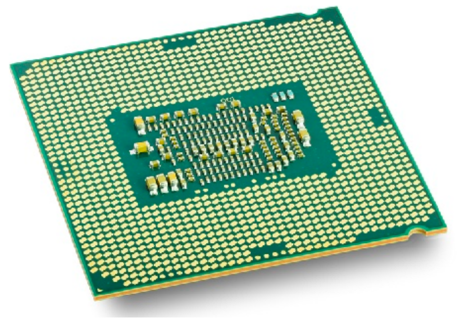
William S. Moses

wsmoses@illinois.edu
MFEM Seminar
Mar 14, 2024


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

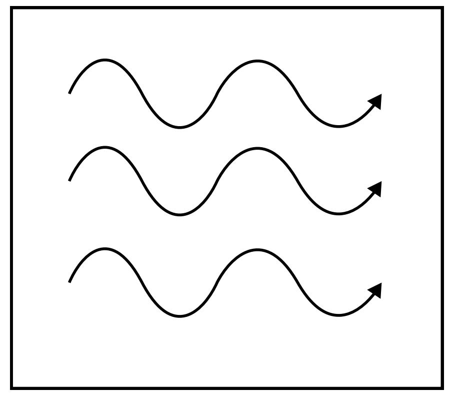
The Programmer's Burden

- 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***;

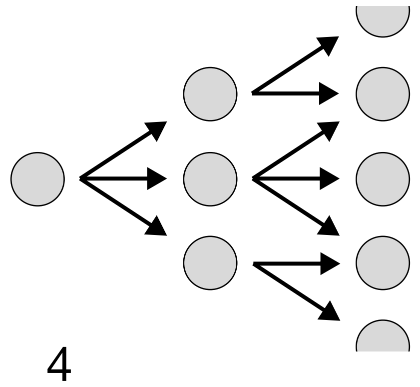


Tapir: understand and optimize ***parallel programs***

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

$A\vec{x}$

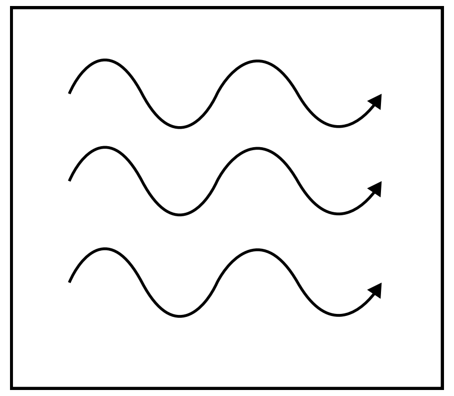
Tensor Comprehensions (TC): automatically ***generate fast tensor arithmetic***



AutoPhase: ***ML-based optimization*** of programs/circuits

Extending the Boundaries of Compilers

Enzyme: fast, parallel, and rewrite-free ***derivative generation***;

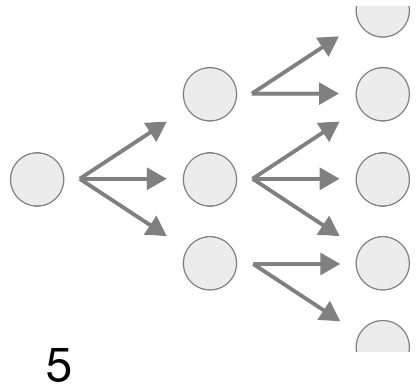


Tapir: understand and optimize ***parallel programs***

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

$A\vec{x}$

Tensor Comprehensions (TC): automatically ***generate fast tensor arithmetic***



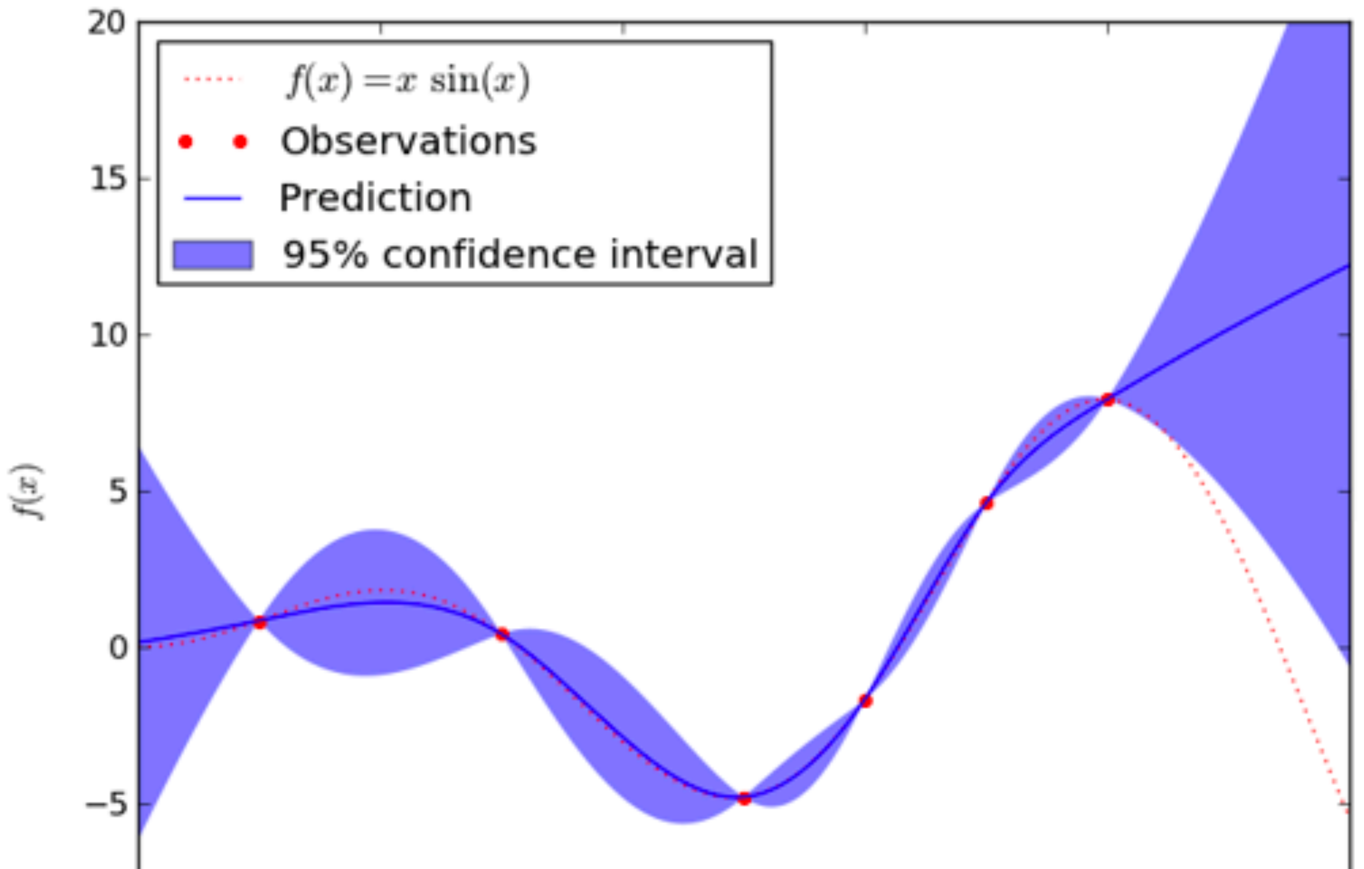
AutoPhase: ***ML-based optimization*** of programs/circuits

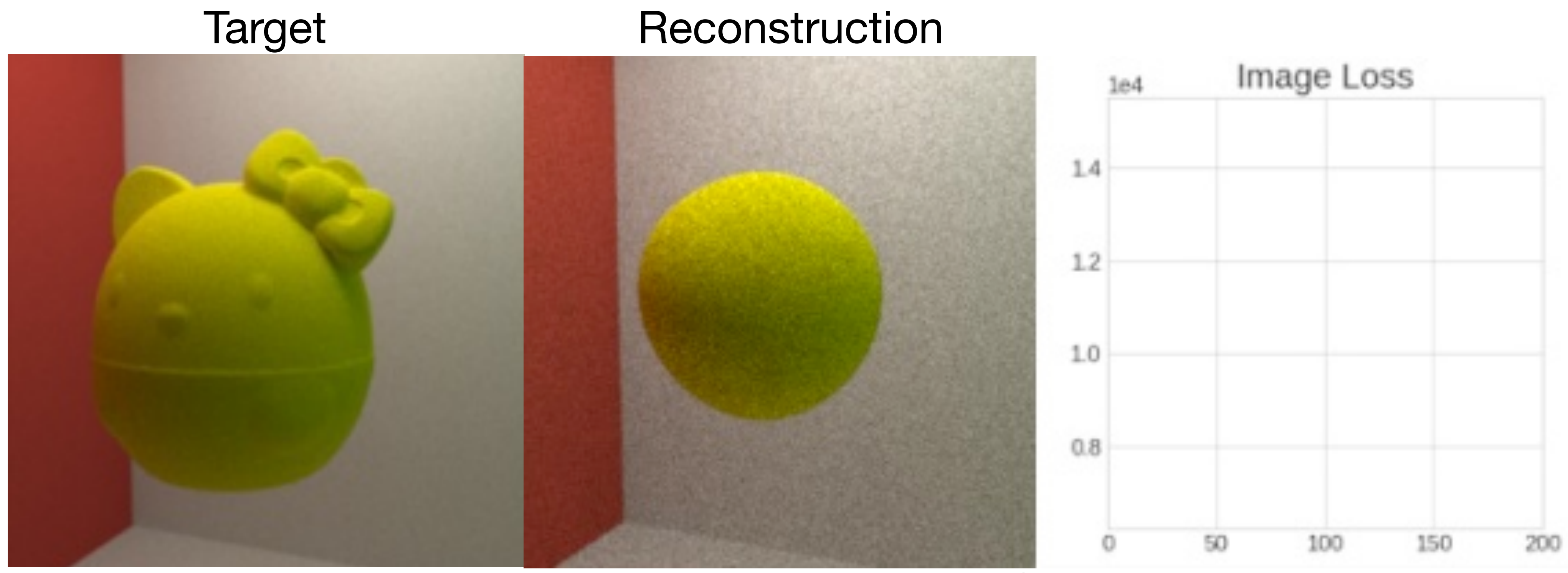
AP Calculus: Revisited

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

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

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



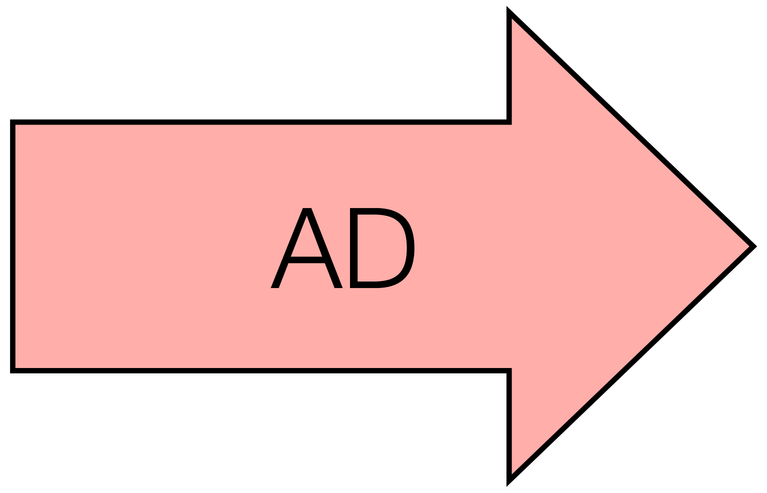


from Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering, SIGGRAPH Asia 2022, Zihan Yu et al

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)

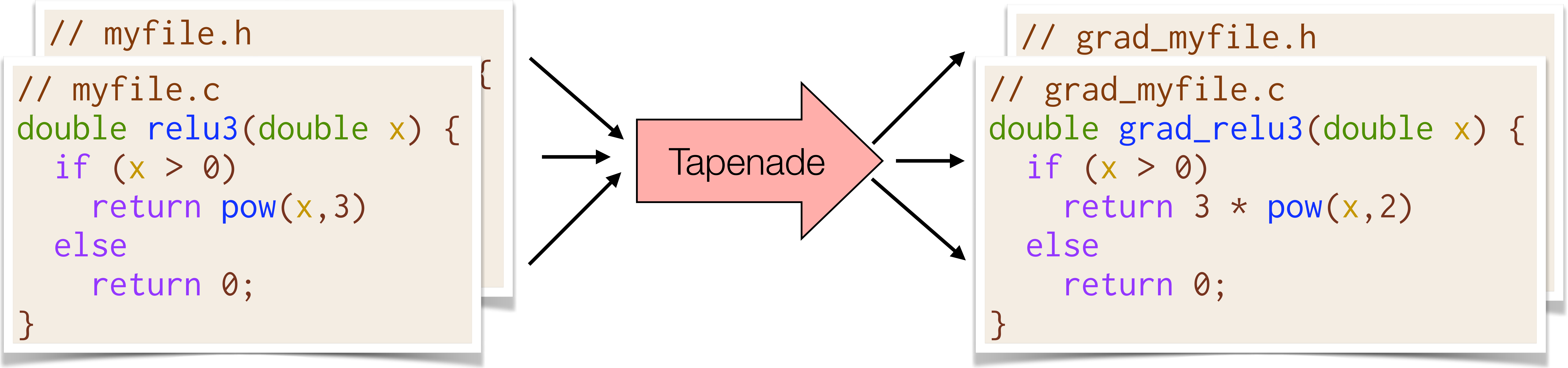
- Operator overloading (Adept, JAX)
  - Differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum)
  - May require writing to use non-standard utilities
  - Often dynamic: storing instructions/values to later be interpreted

```
// Rewrite to accept either
//    double or adouble
template<typename T>
T relu3(T val) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```

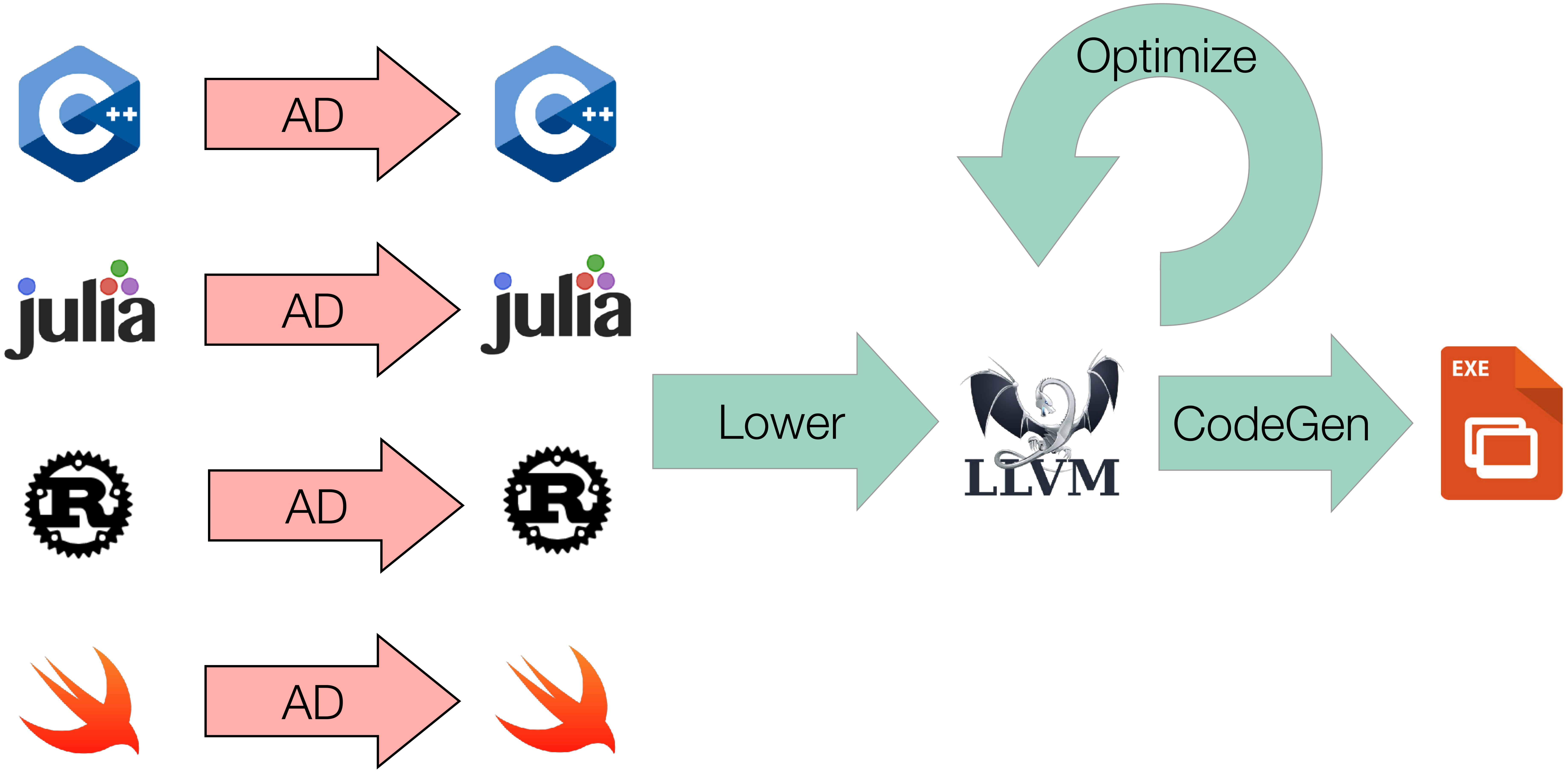
```
adept::Stack stack;
adept::adouble inp = 3.14;

// 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

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

Optimization & Automatic Differentiation

$O(n^2)$

```
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

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

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

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

AD

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

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

Optimize

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

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

Optimization & Automatic Differentiation

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

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

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

Optimize →

$O\left(n\right)$

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

AD →

$O\left(n\right)$

```
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]…
}
∇mag(d_in, d_res)
```

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

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

AD →

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

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

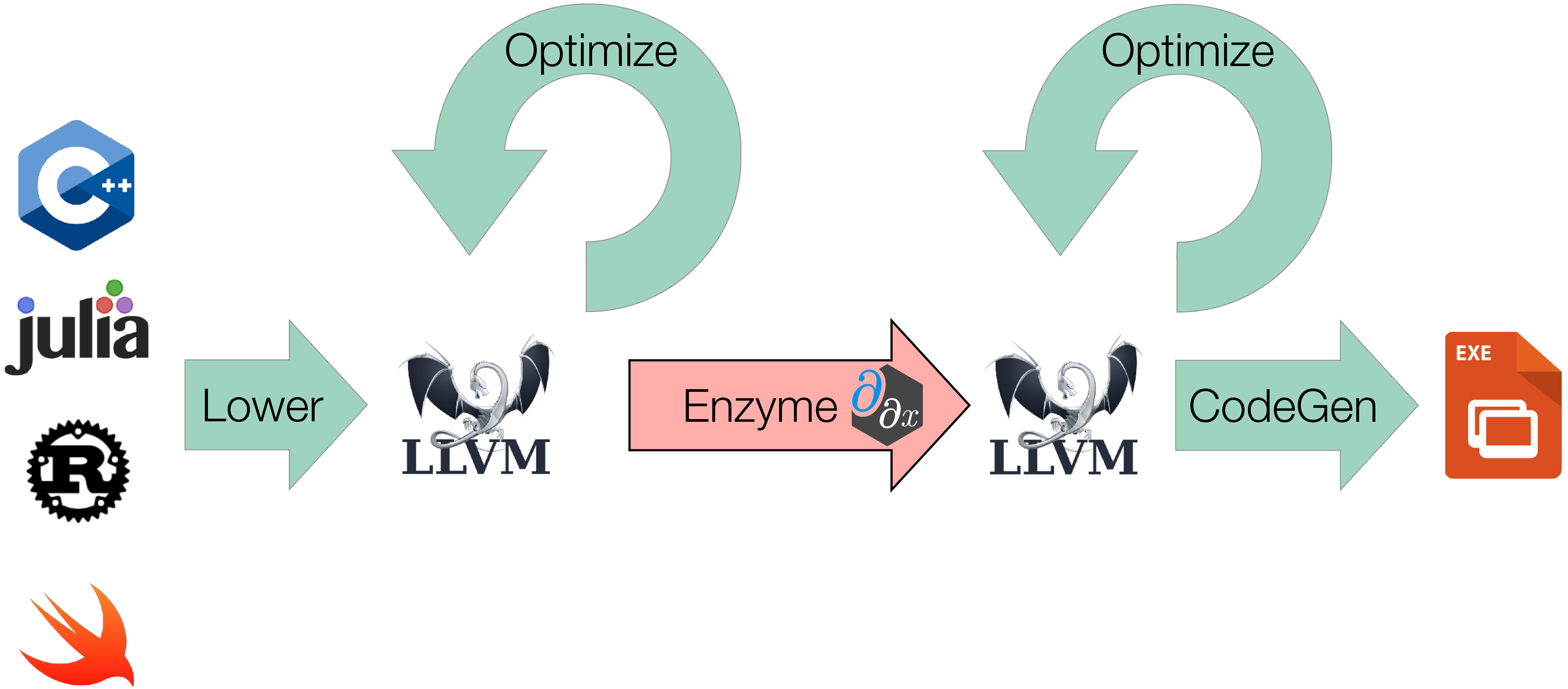
Optimize →

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

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

Enzyme Approach

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

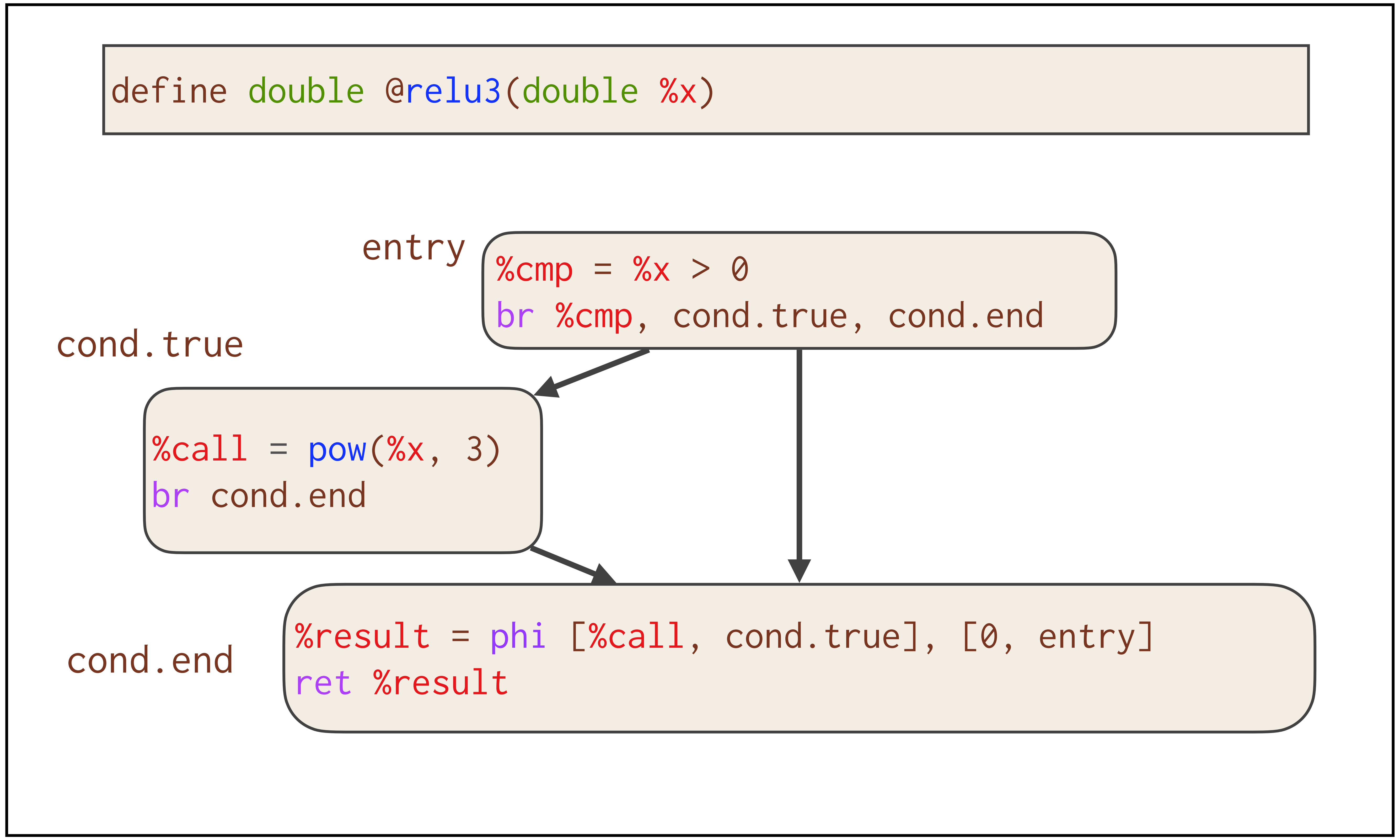


Case Study: ReLU3

C Source

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

LLVM

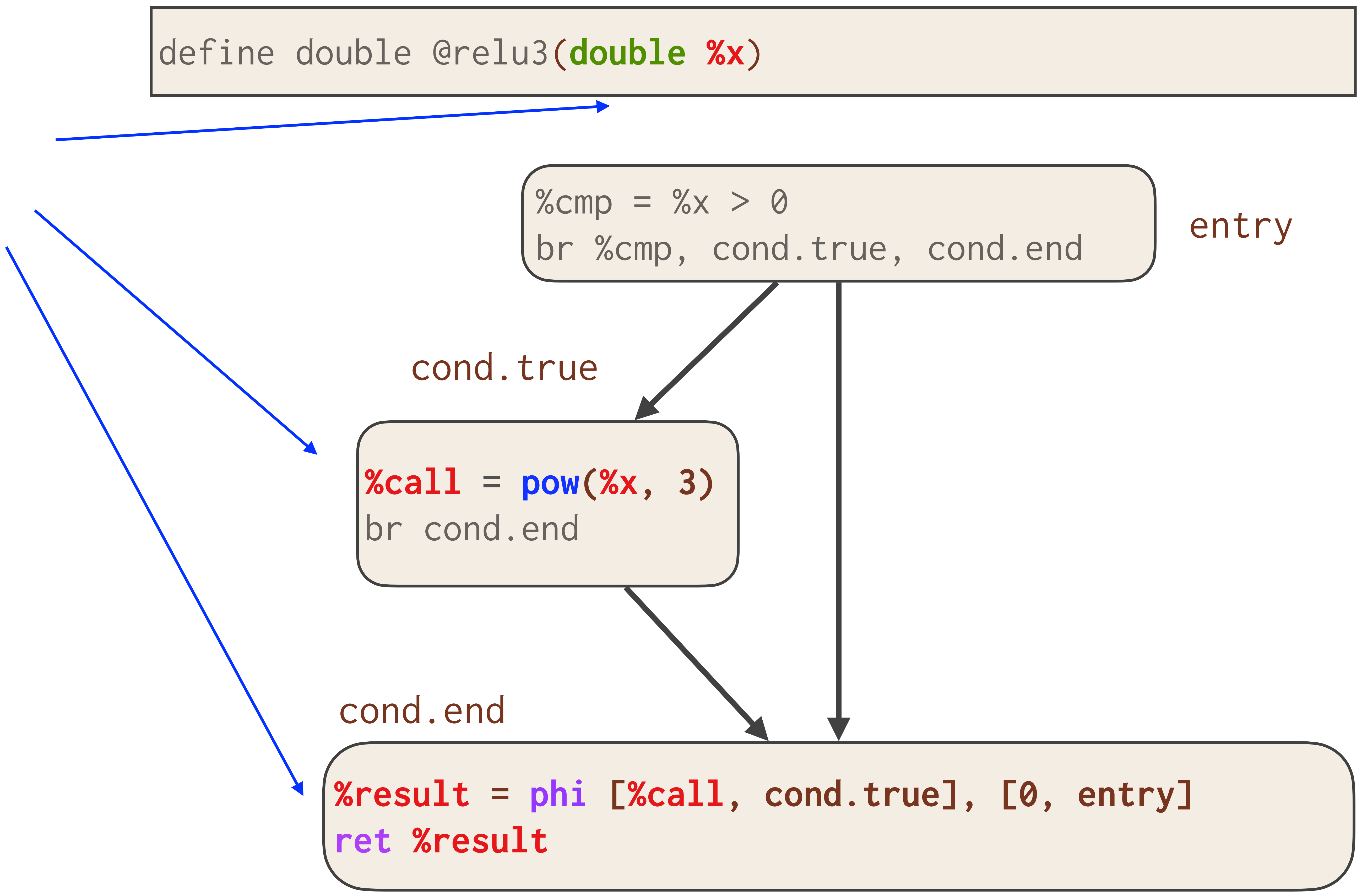


Enzyme Usage

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

Case Study: ReLU3

Active Instructions



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

entry

```
alloca %result' = 0.0
alloca %call'   = 0.0
alloca %x'      = 0.0
%cmp = %x > 0
br %cmp, cond.true, cond.end
```

Allocate & zero shadow memory for active values

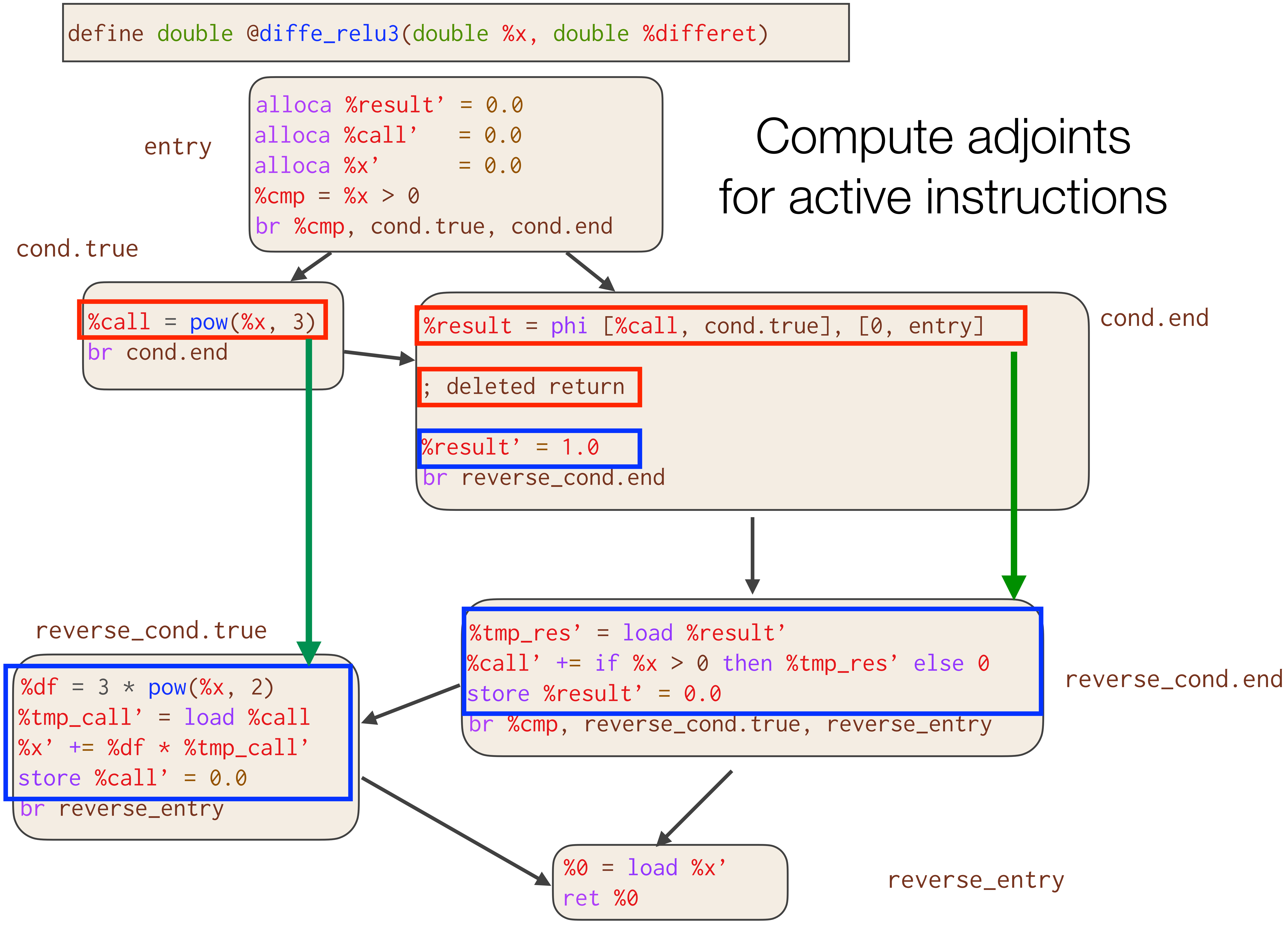
cond.true

```
%call = pow(%x, 3)
br cond.end
```

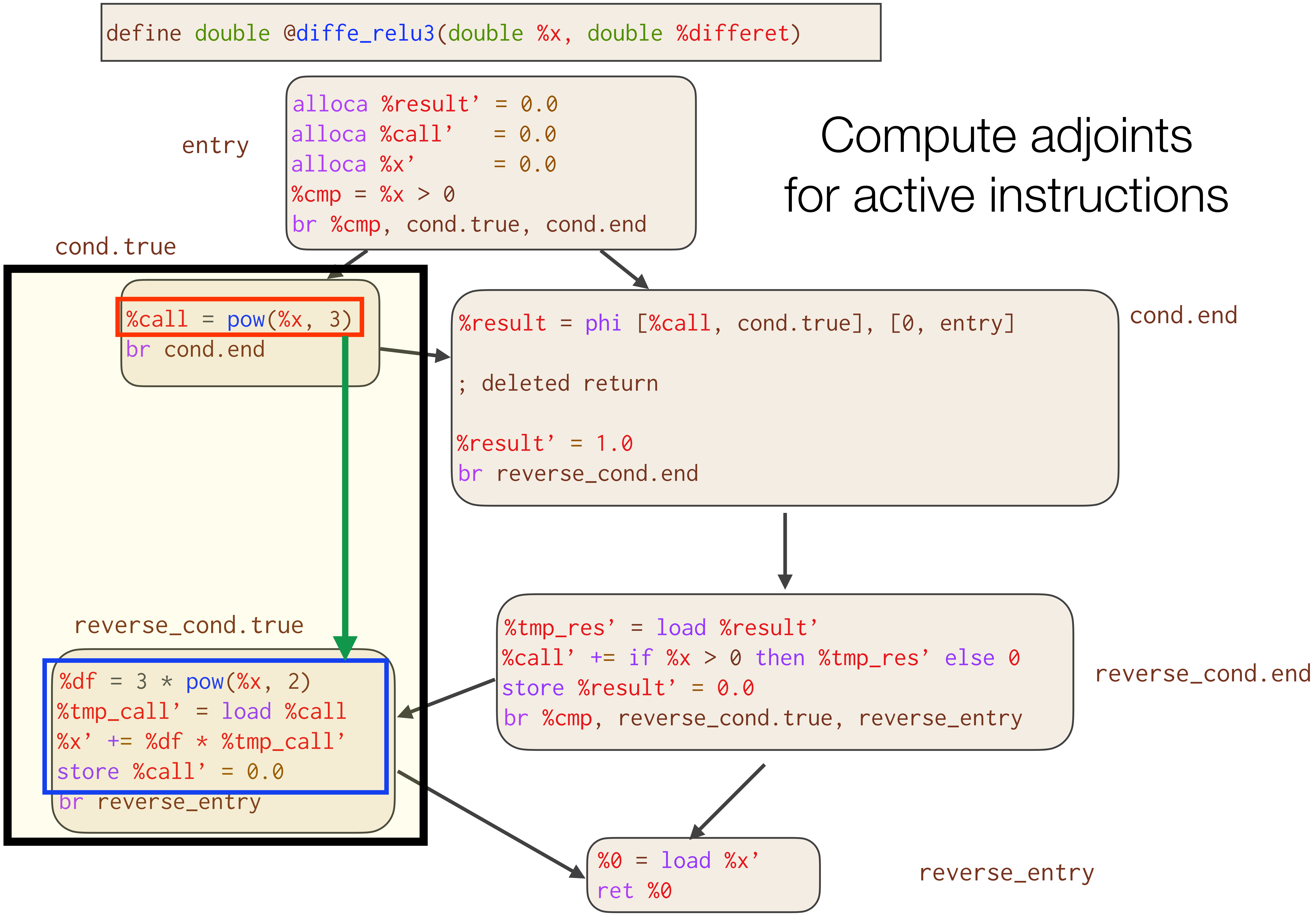
cond.end

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

; deleted return

%result' = 1.0
br reverse_cond.end
```

Compute adjoints for active instructions



Compute adjoints for active instructions



Post Optimization

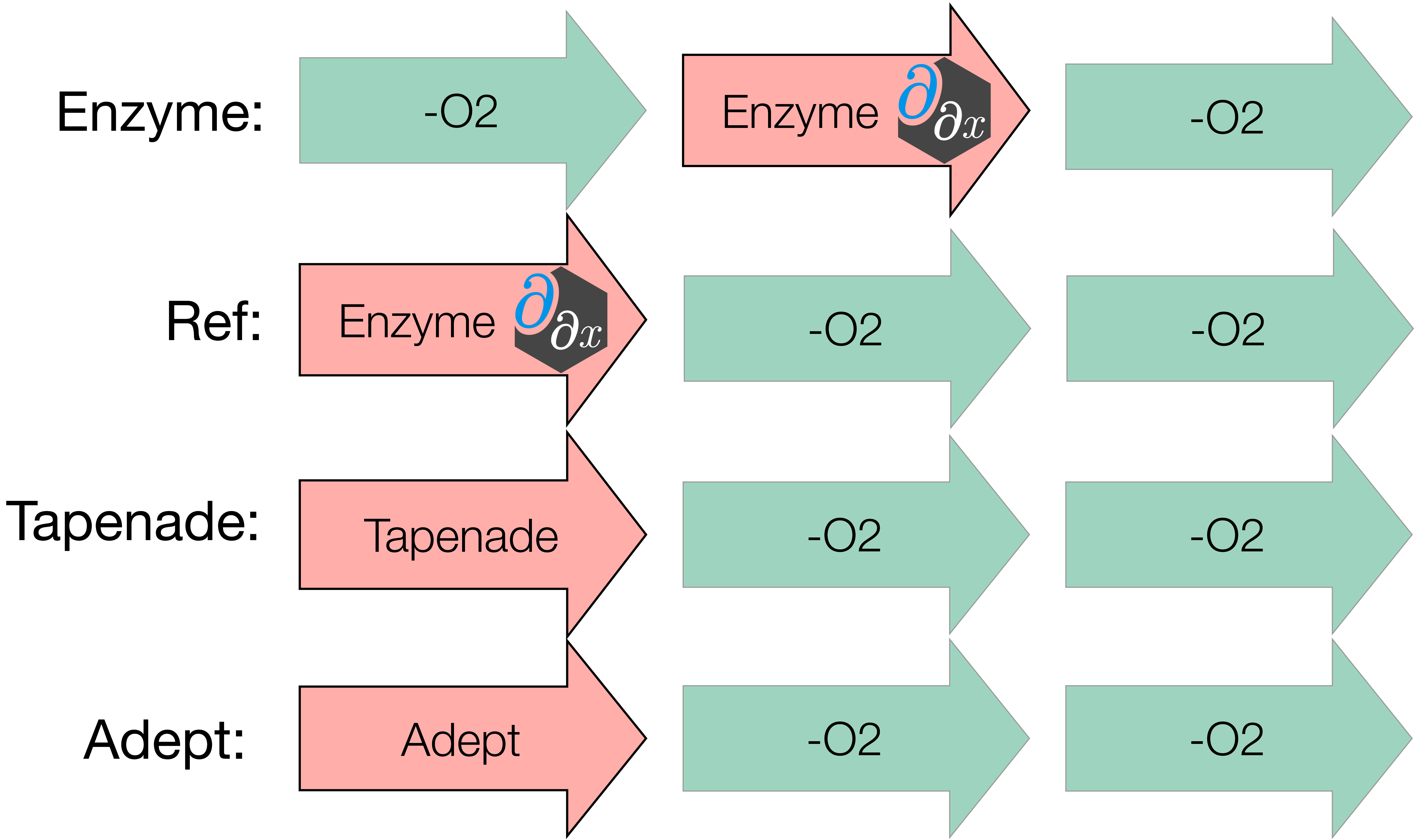
```
define double @diffe_relu3(double %x)

entry:
  %cmp = %x > 0
  br %cmp, reverse_cond.true, reverse_entry

reverse_cond.true:
  %3 = 3 * pow(%x, 2)
  br reverse_entry

reverse_entry:
  %0 = phi [%3, reverse_cond.true], [0, entry]
  ret %0
```

Essentially the optimal hand-written gradient!

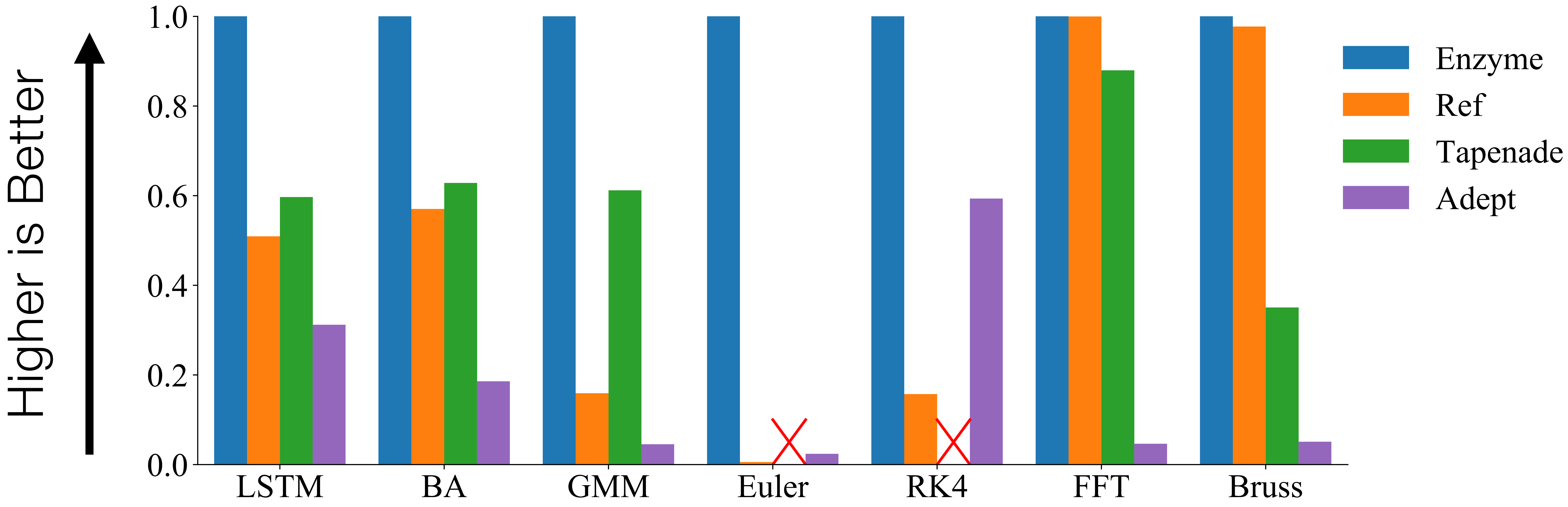
```c
double diffe_relu3(double x) {
  double result;
  if (x > 0)
    result = 3 * pow(x, 2);
  else
    result = 0;
  return result;
}
```

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;
```

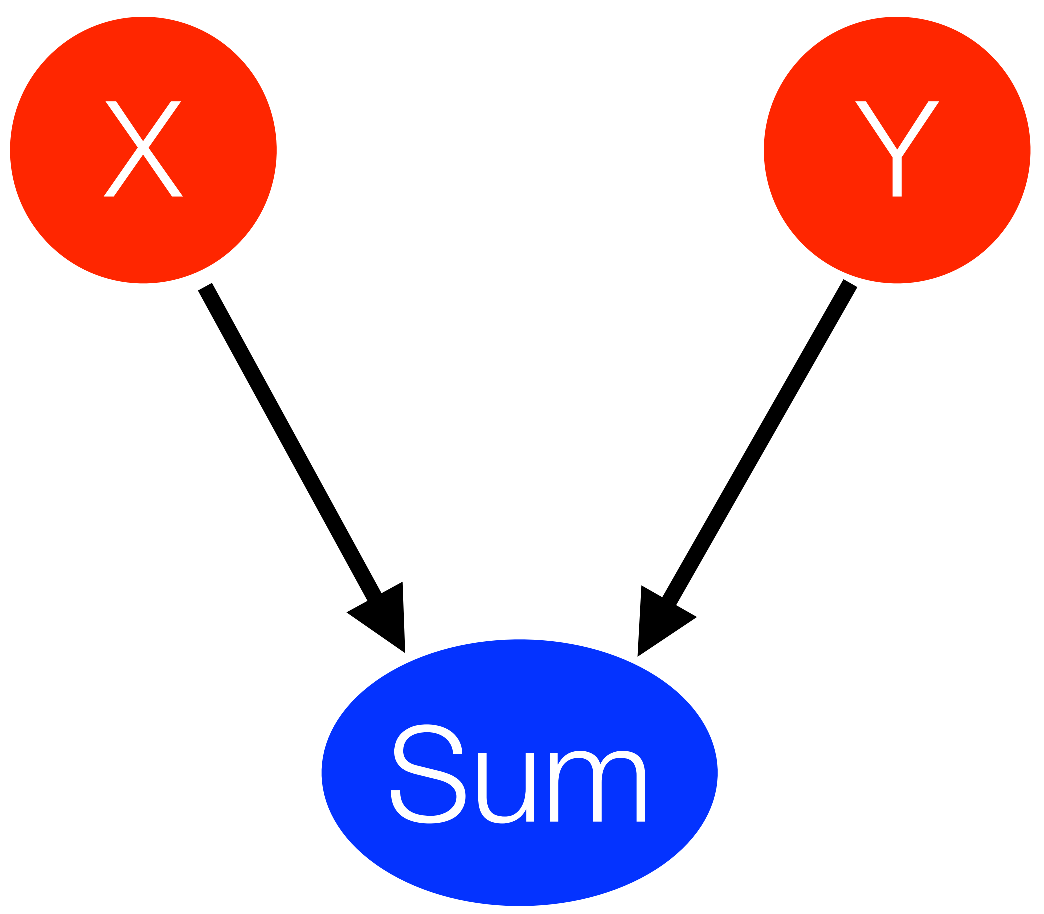
Cache Reduction Example

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

```
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);
}
```

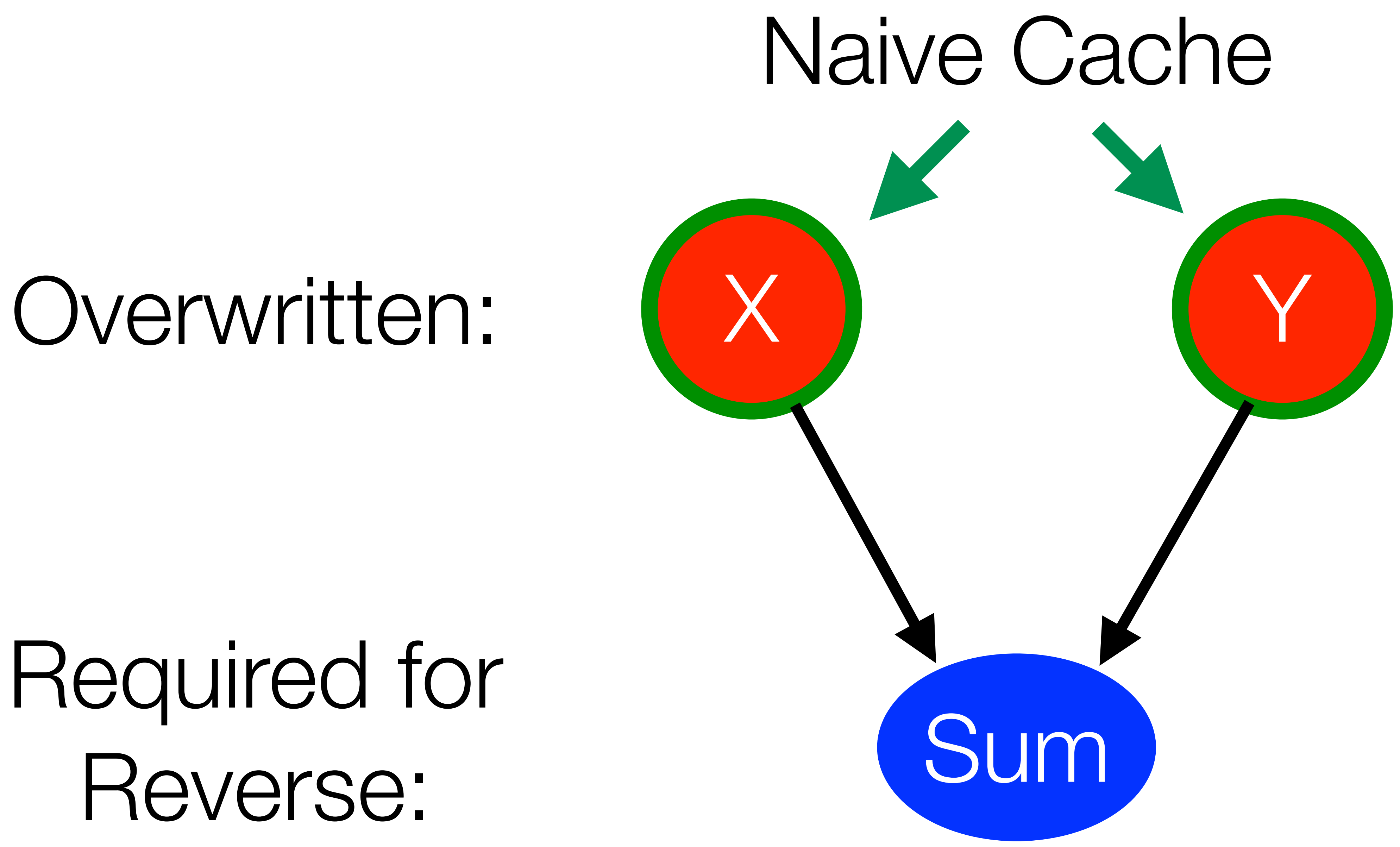
Overwritten:

Required for Reverse:



Cache Reduction Example

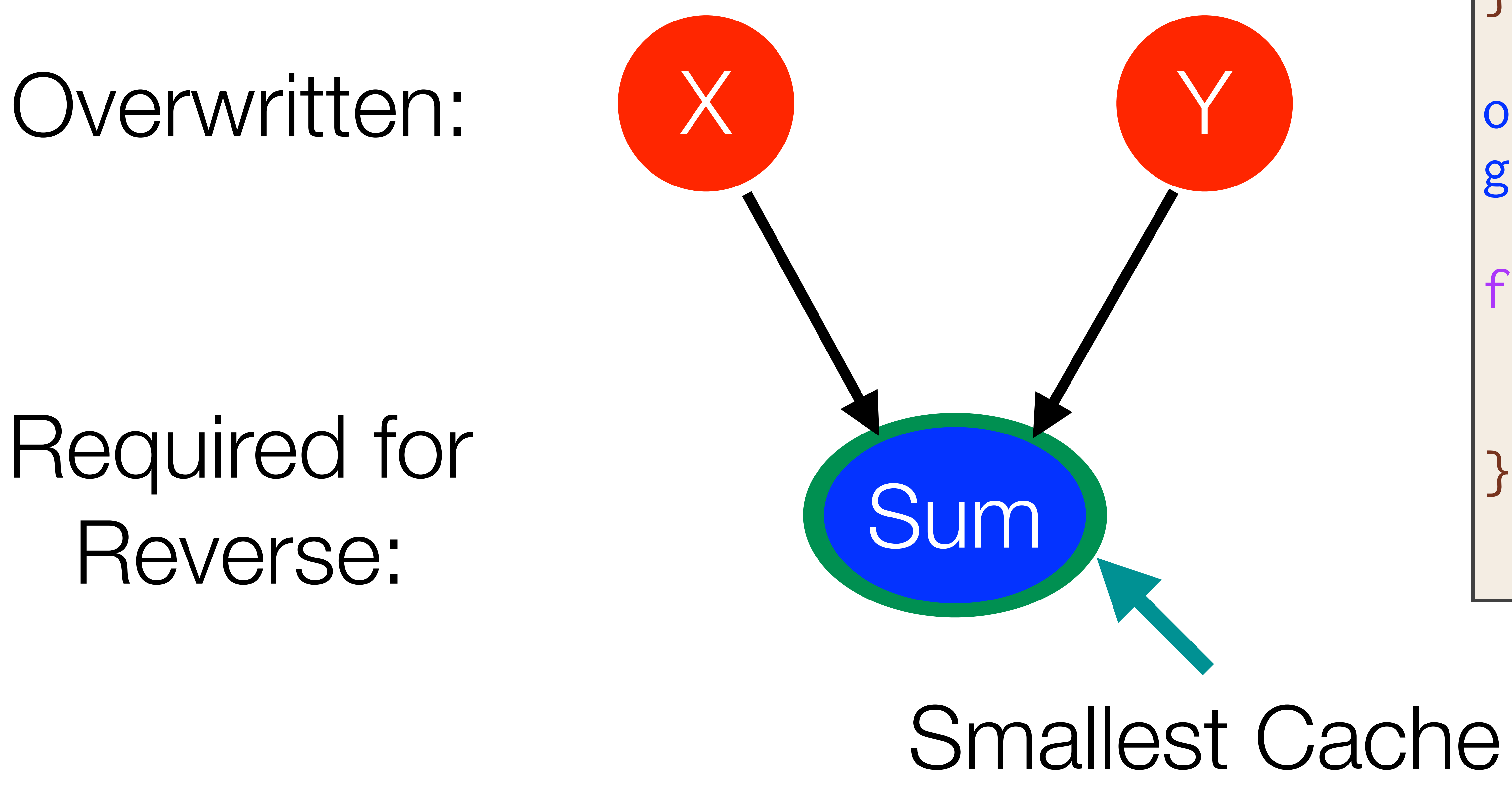
- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.




```
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.



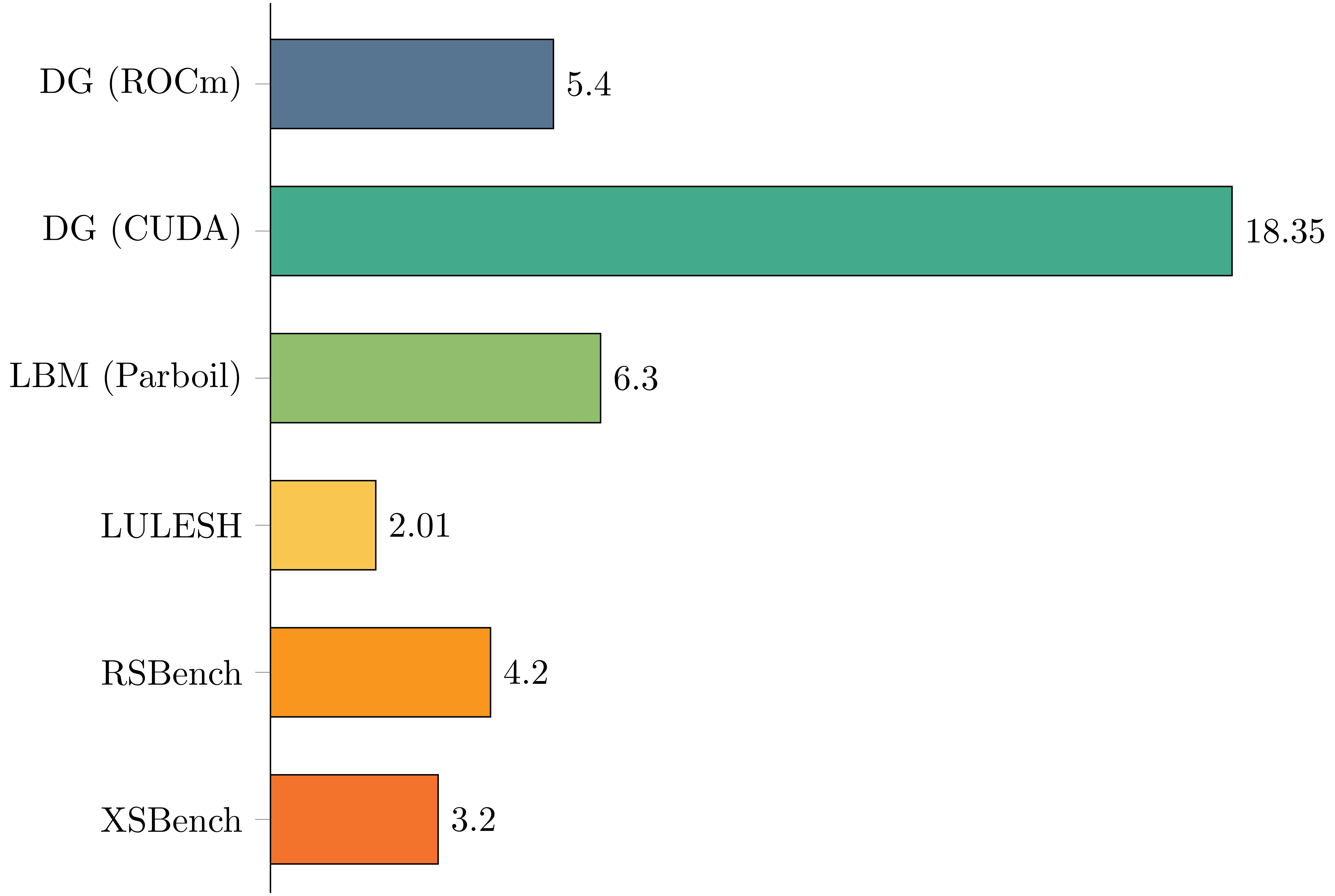

```
double* sum_cache = new double[10];

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]);
}
```

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
- …

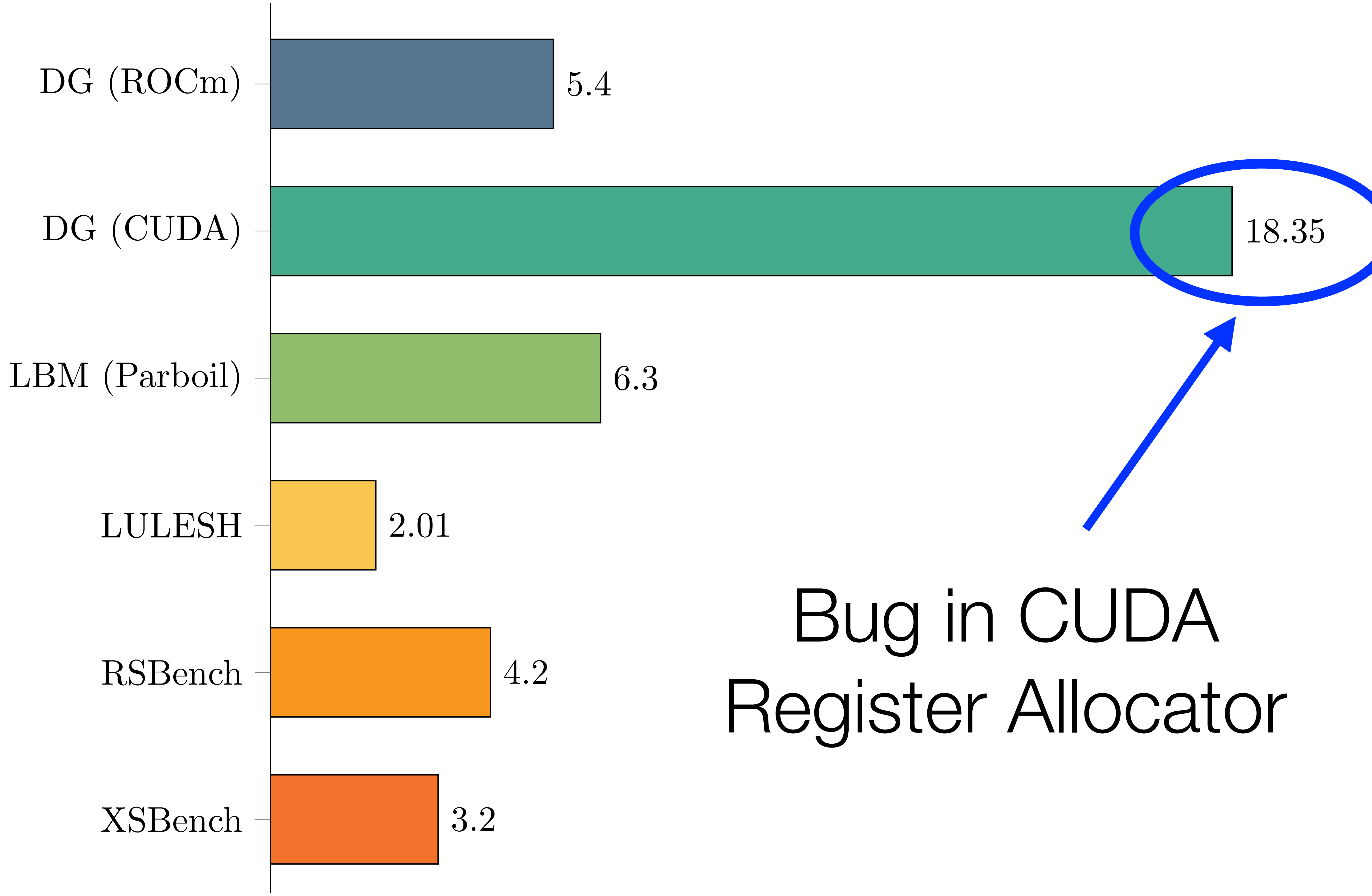
GPU Gradient Overhead

- 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)



Ablation Analysis of Optimizations

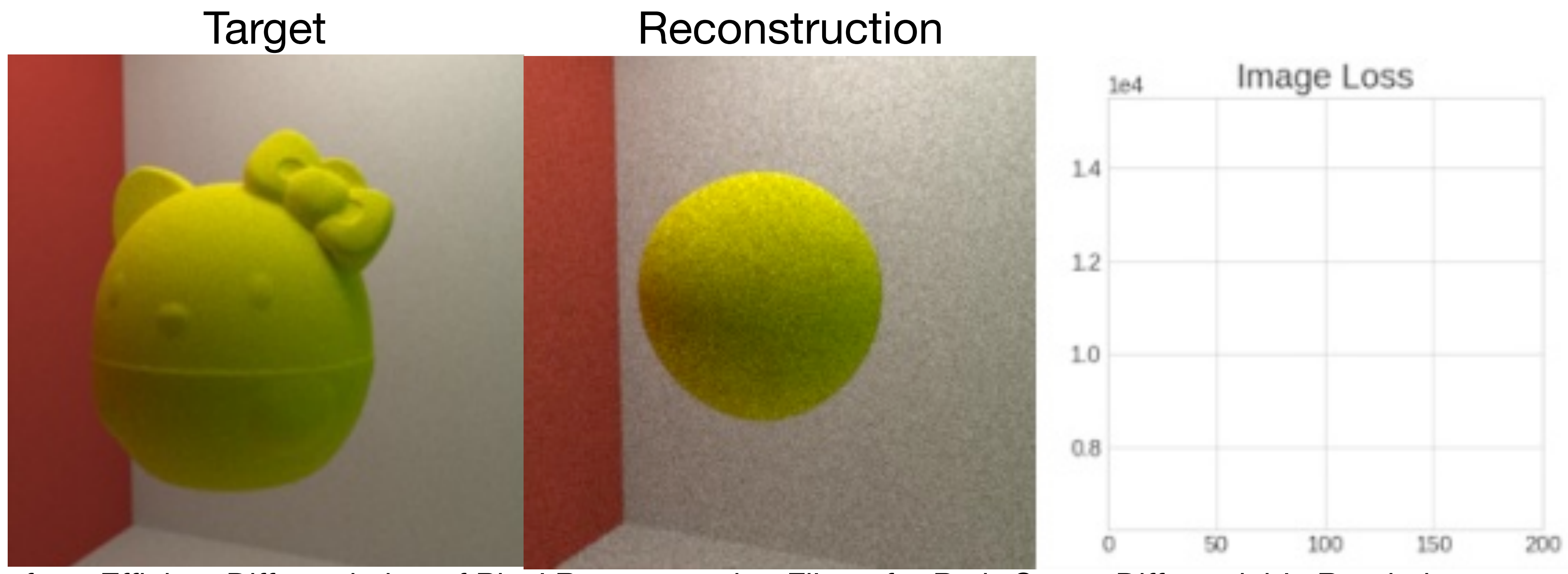
Ablation Analysis of Optimizations

Ablation Analysis of Optimizations

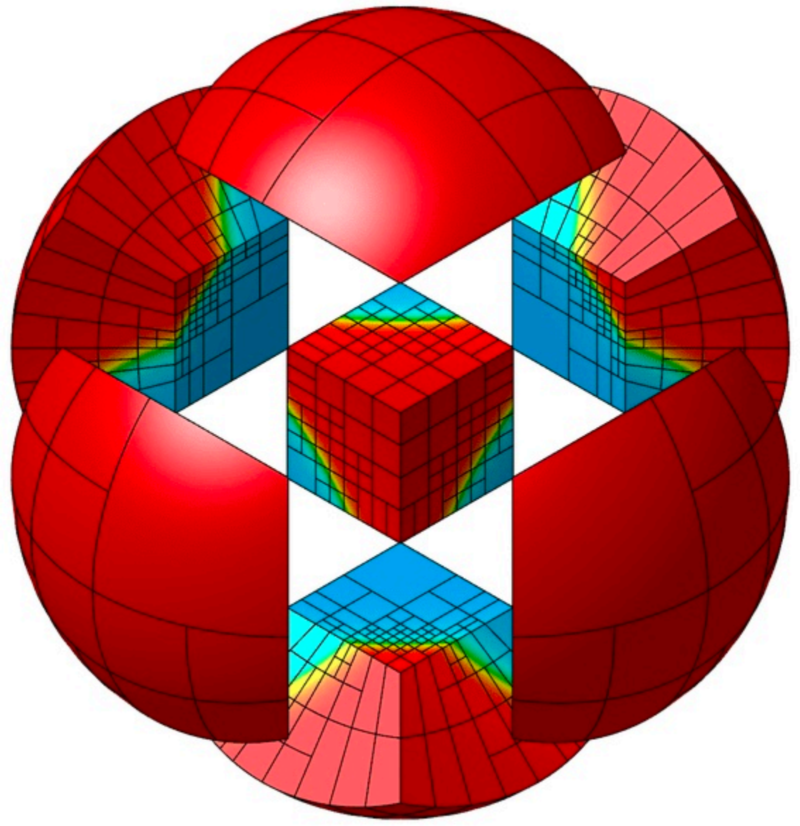
Ablation Analysis of Optimizations

GPU AD is Intractable Without Optimization!

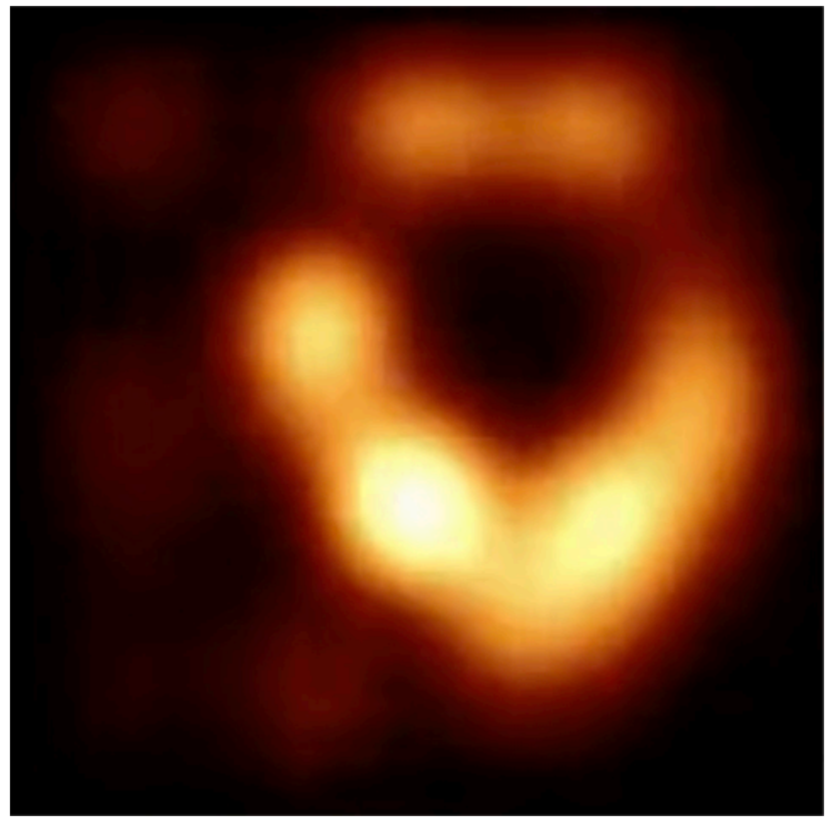
Enzyme-Powered Applications



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



from MFEM Team at LLNL



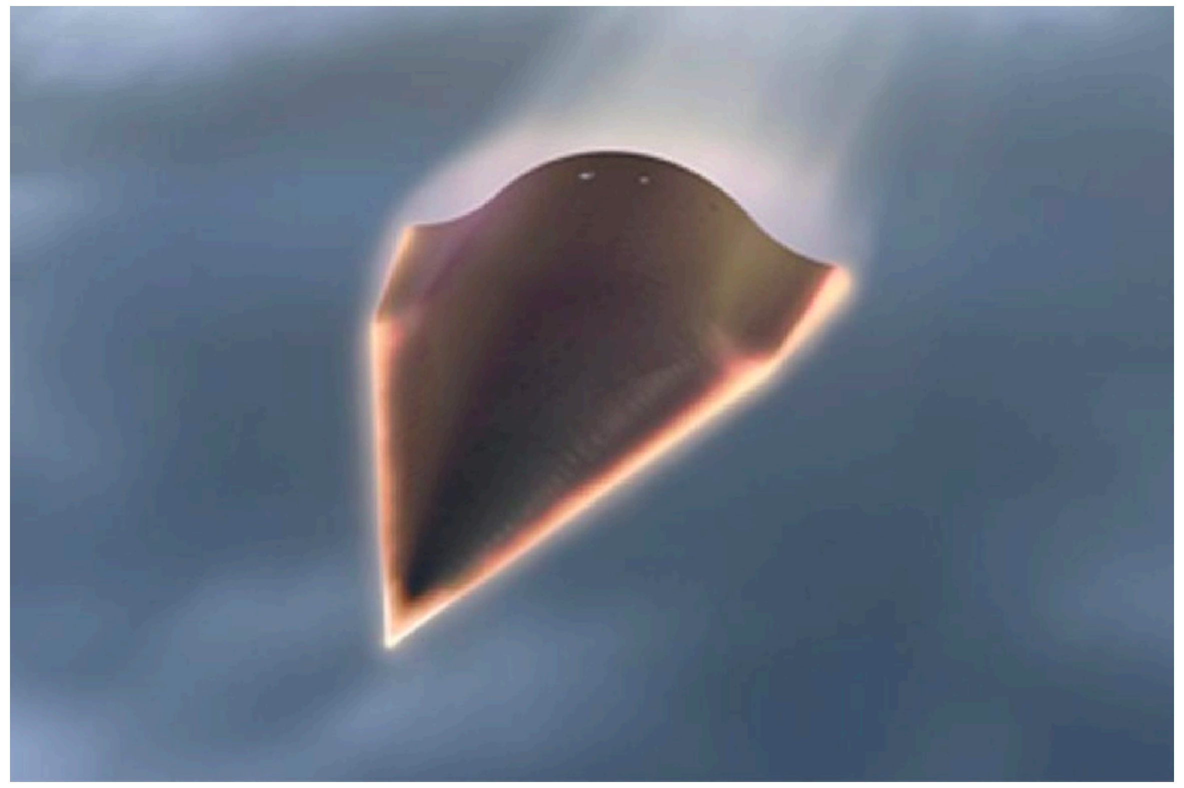
>100x speedup!

Prior:
5 days (cluster)

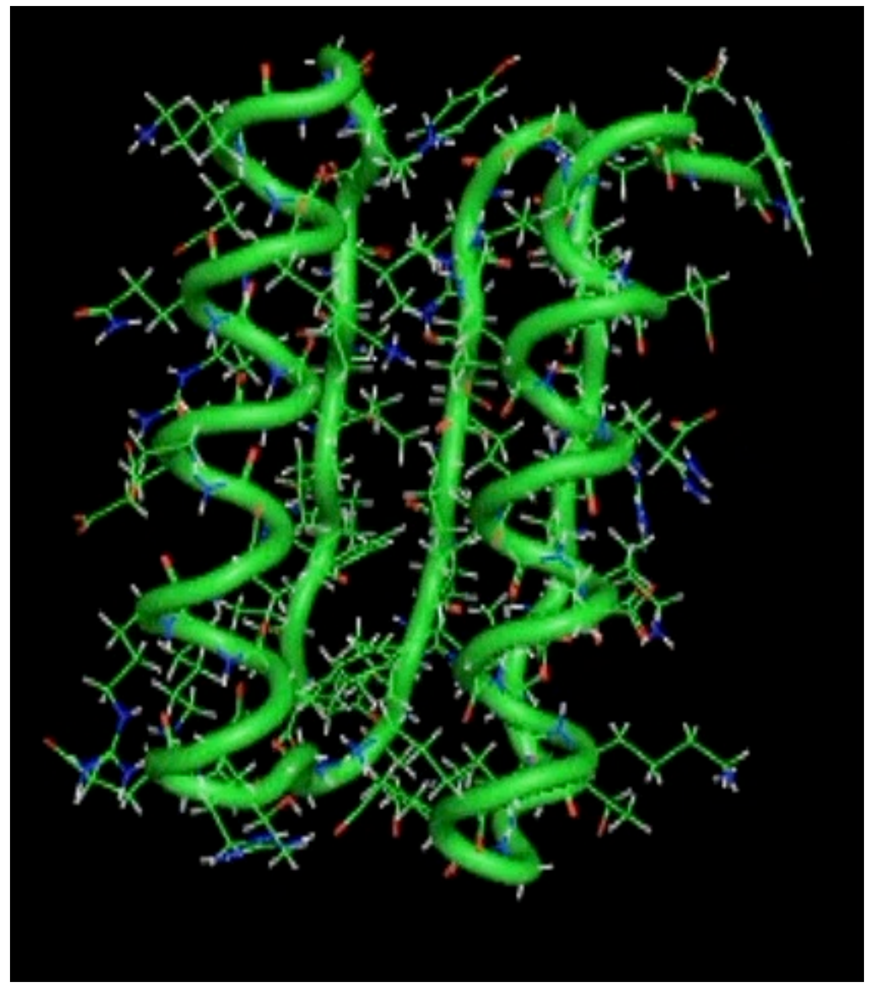
Enzyme-Based:
1 hour (laptop)

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)



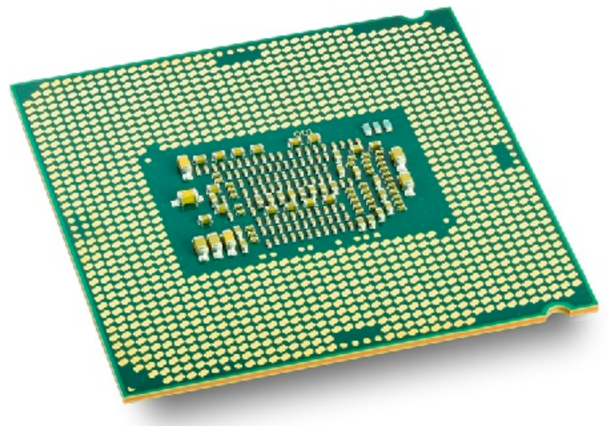
from Center for the Exascale Simulation of Materials in Extreme Environments



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);
  }
}
```

A parallel loop replaces the original serial loop

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);
  }
}
```

A parallel loop replaces the original serial loop

N = 64M

Serial Running time: 0.312 s

18-core Running time: 180.657s

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);
  }
}
```

A parallel loop replaces the original serial loop

N = 64M

Serial Running time: 0.312 s

18-core Running time: 180.657s

1-core Running time: 2600.287s

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);
}
```

X

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

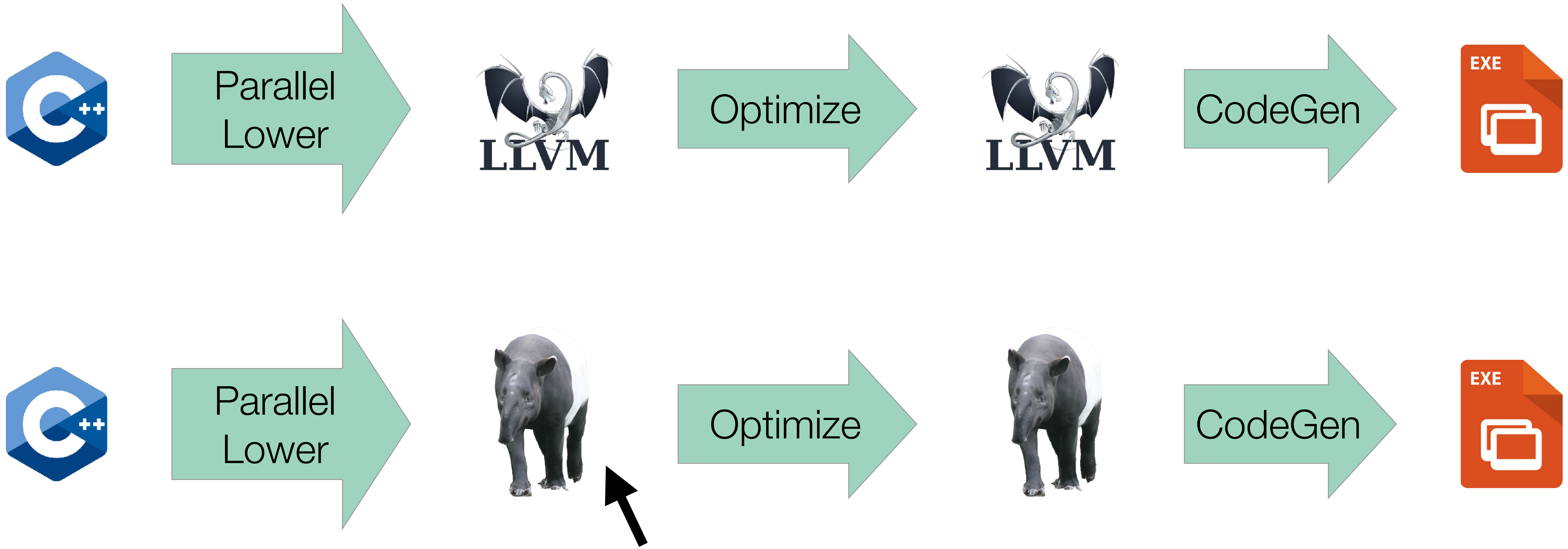
Compiling Parallel Code (Realistic)

```
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



New IR that encodes parallelism for optimization!

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]

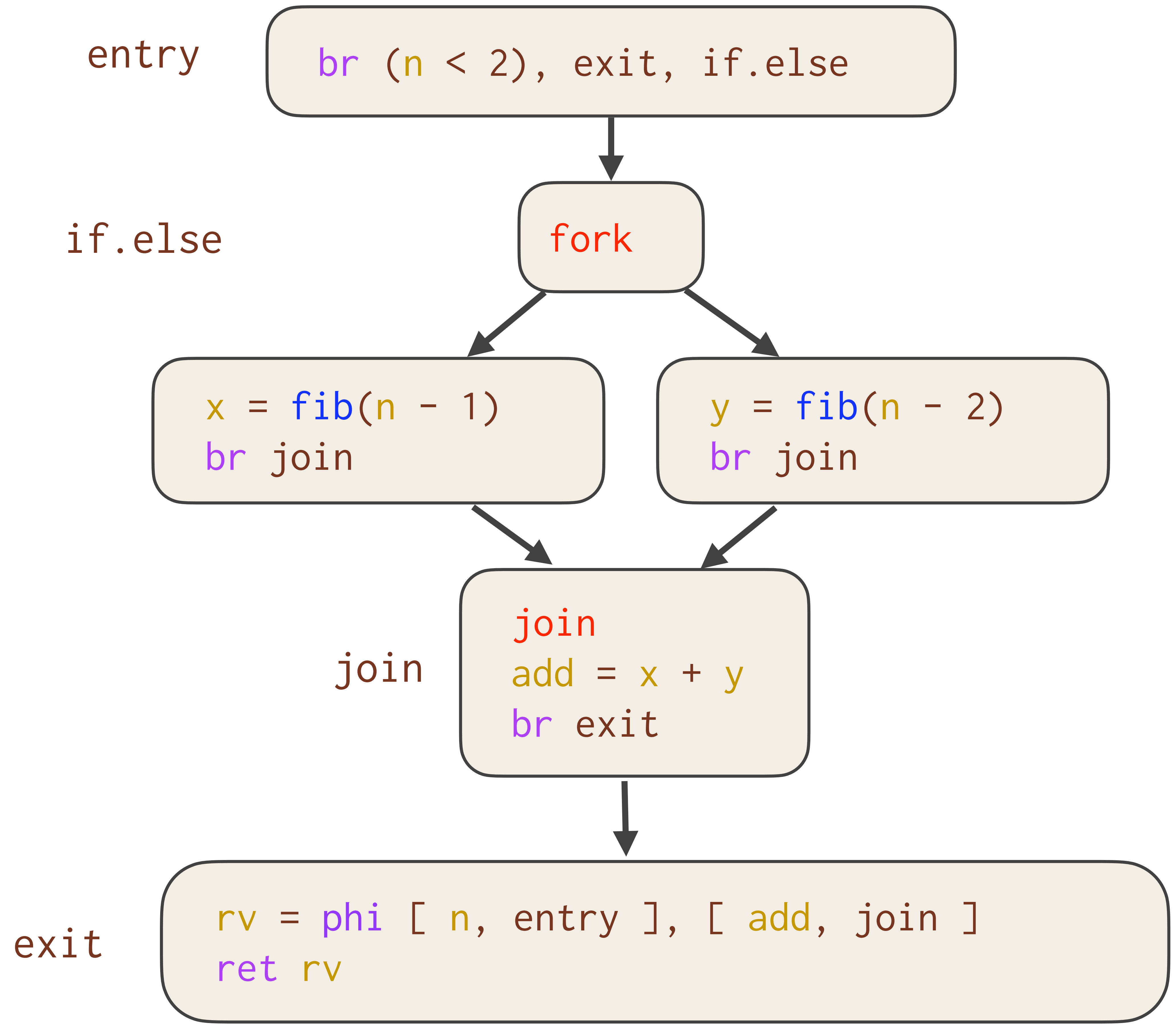
Example Previous Parallel IR

- 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;
}
```

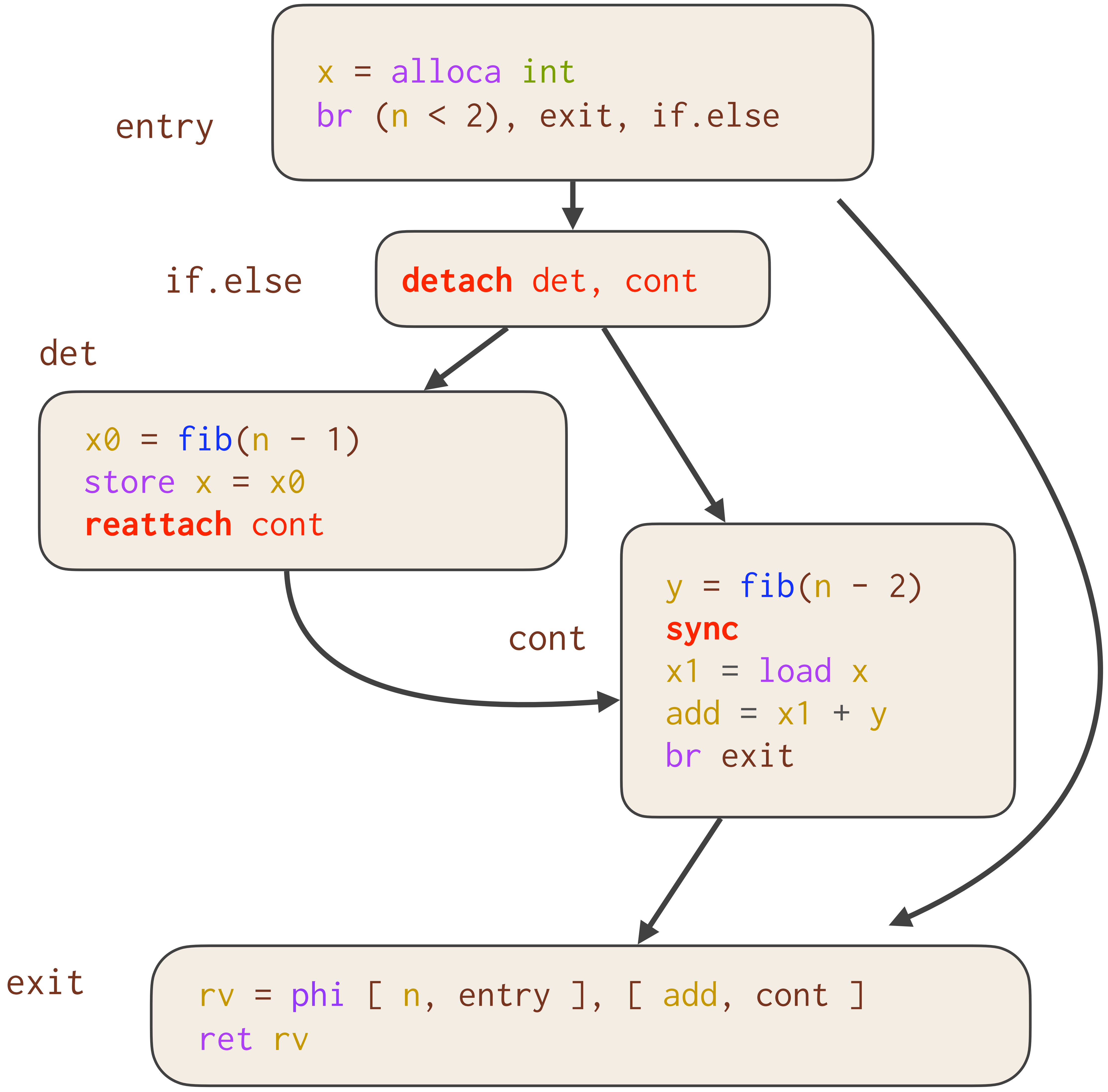
Problem: The `join` block **breaks implicit assumptions** made by the compiler.

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



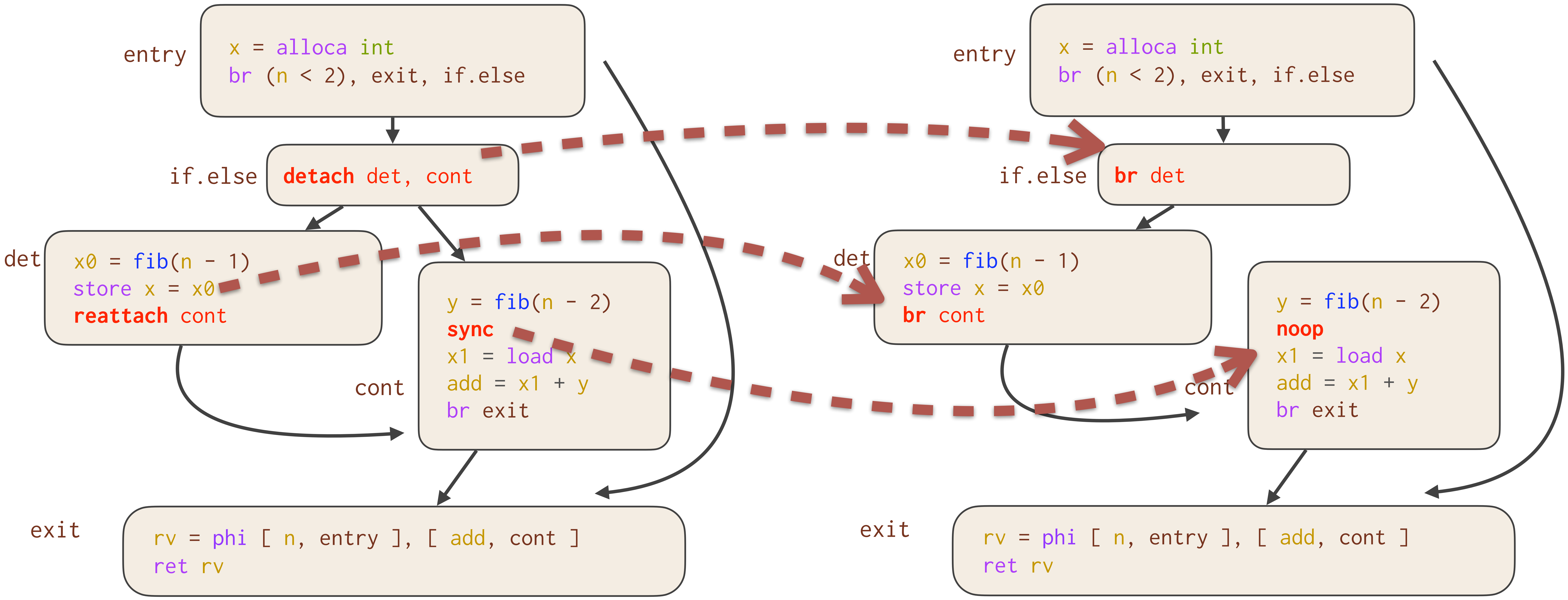
Tapir: Task-Based Asymmetric Parallel IR

- 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

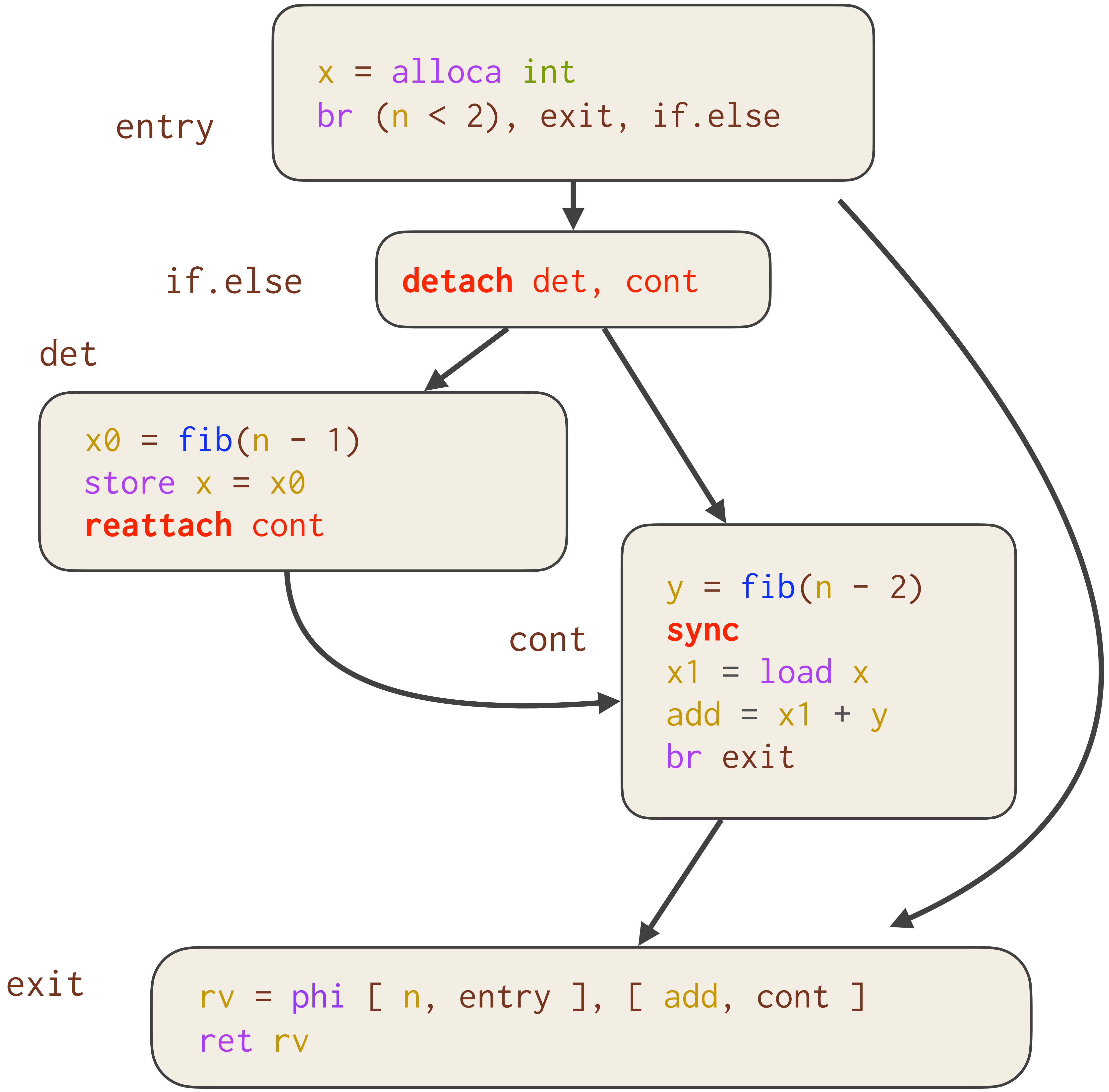
- Reasoning about parallelism is a minor change to reasoning about the **serial projection**.



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**.

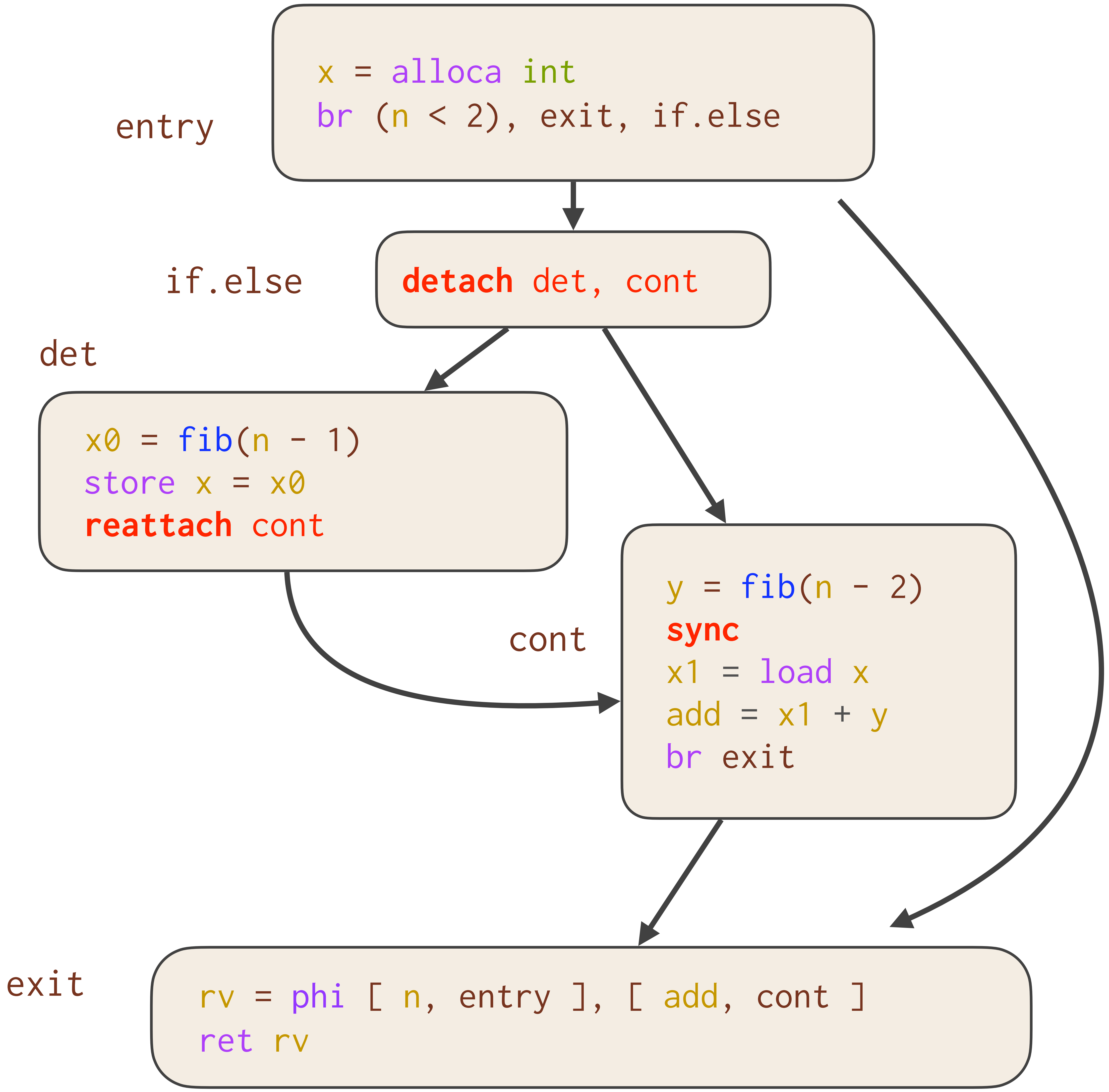


Maintaining Correctness

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

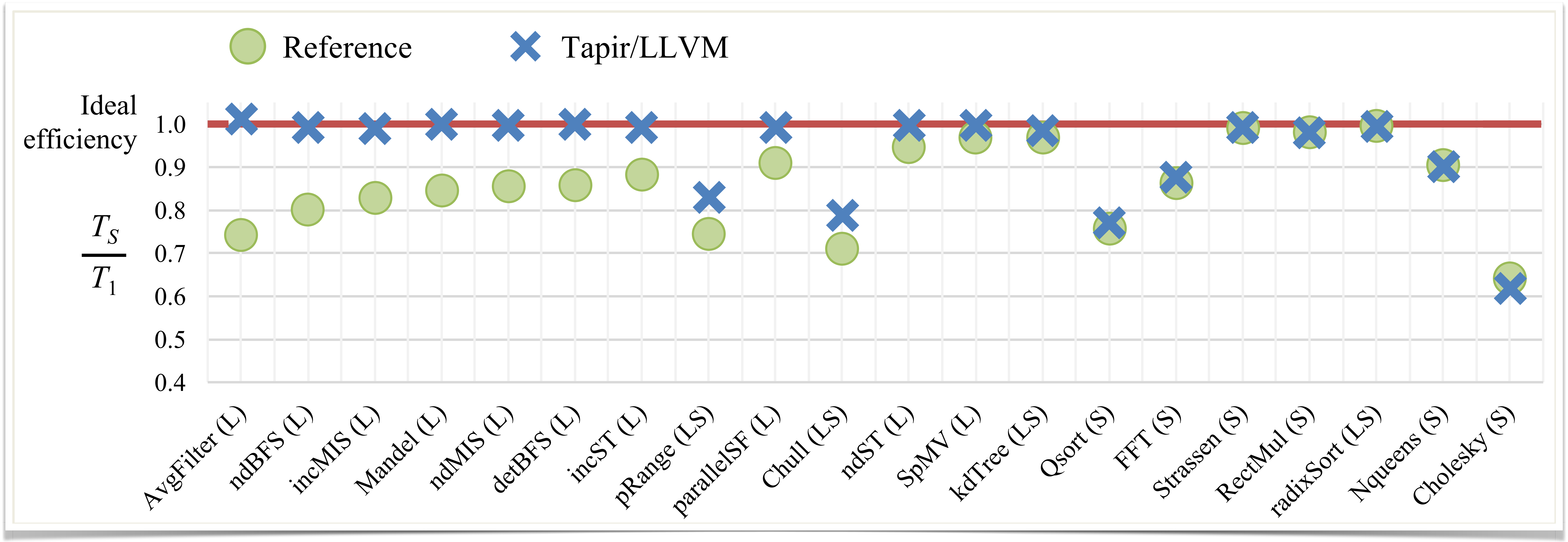
N = 64M

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



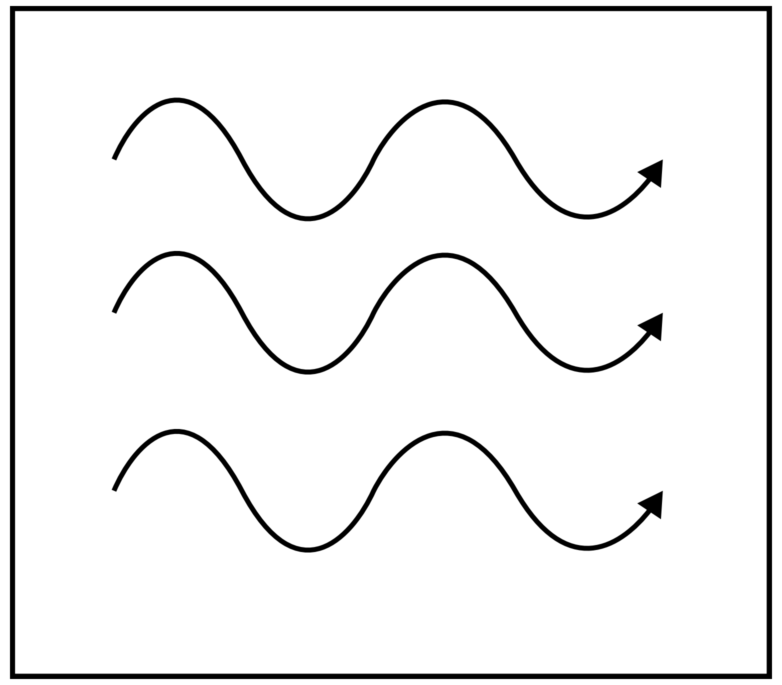
Decreasing difference between Tapir / LLVM and Reference

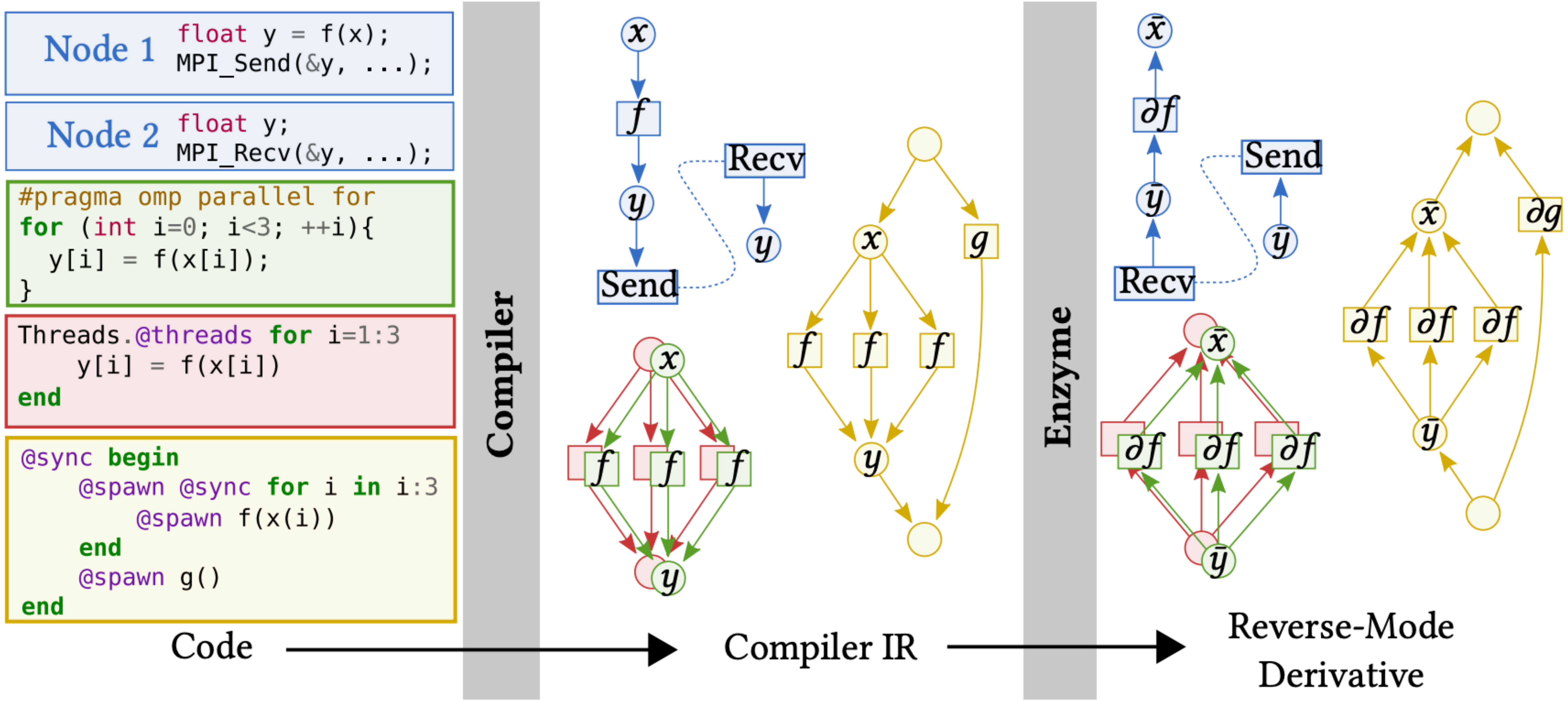
Polygeist: Extending Parallel IRs beyond Multicore

- Good IR representations are especially necessary for device-specific constructs, like GPU syncthreads
  - Necessary for good performance, but complexity means they're often used poorly
  - General abstracts can enable code written in one framework to be used ***and high-performance*** on many others without rewriting
  - Recompiled PyTorch's GPU backend to produce an efficient CPU backend that runs 2.7x faster than PyTorch's native CPU code!

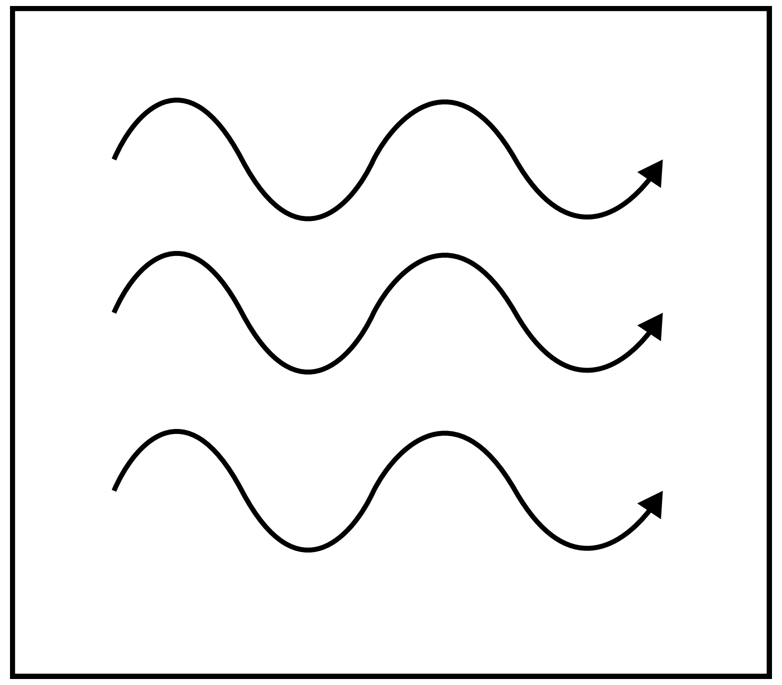
```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];

  if ( tx == 0 )
    node[ty] = input[index_in] ;

  // Unnecessary Barrier #1
  // None of the read/writes below the sync
  //  (weights, hidden)
  // intersect with the read/writes above the sync
  //  (node, input)
  __syncthreads();


  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];

  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  …
}
```

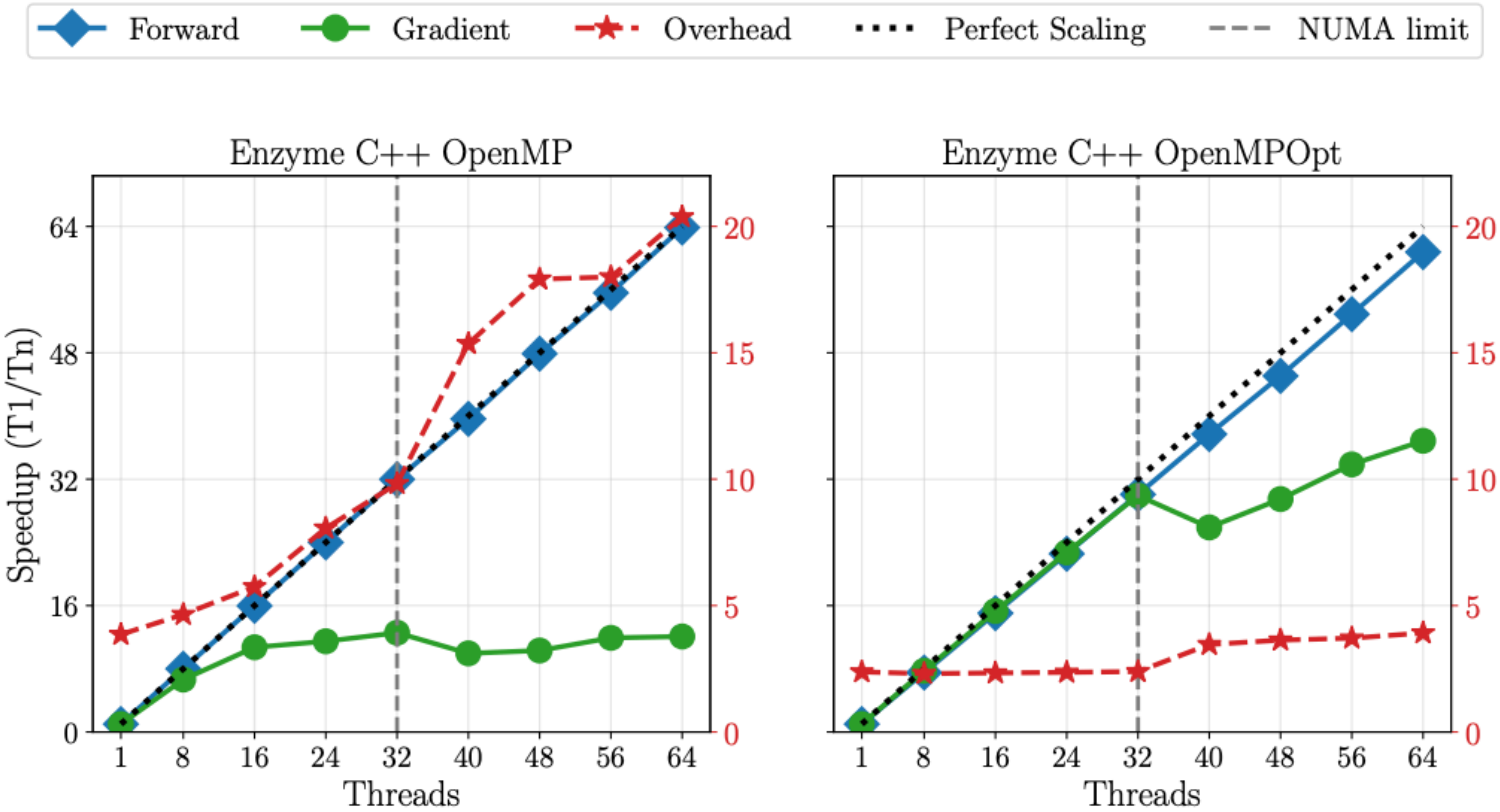
Revisiting The Programmer's Burden





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. Polygeist extends these ideas to GPU programs and enables write-once run-anywhere.

- 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

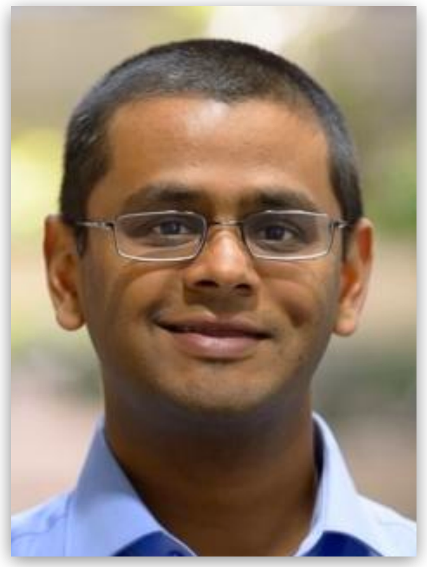

Jan Hückelheim


Charles E. Leiserson


Zach Devito

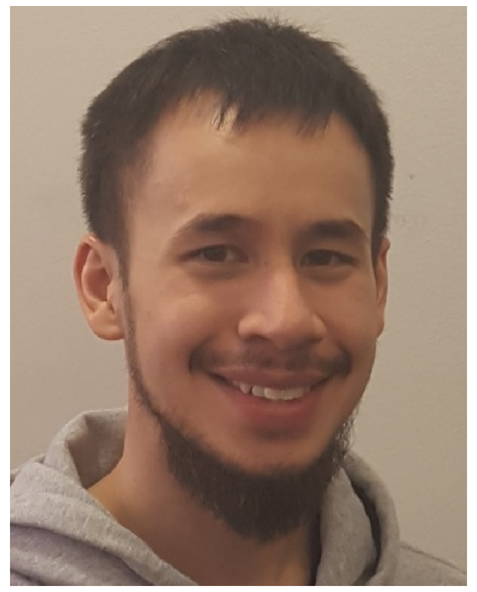

Andrew Adams


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

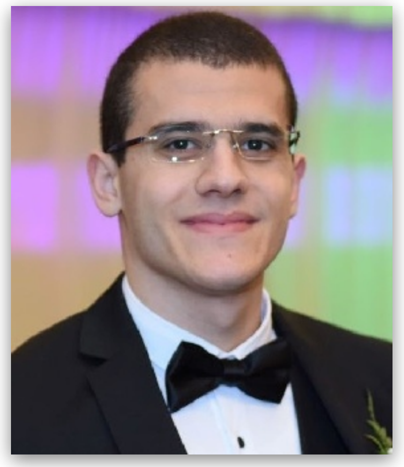

Theodoros Theodoridis


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