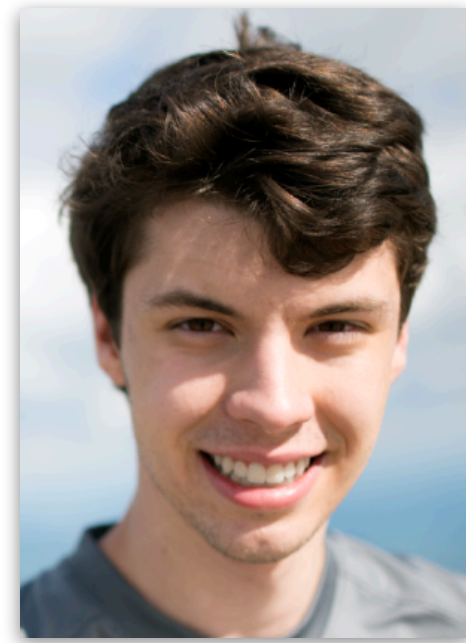




Enzyme.jl

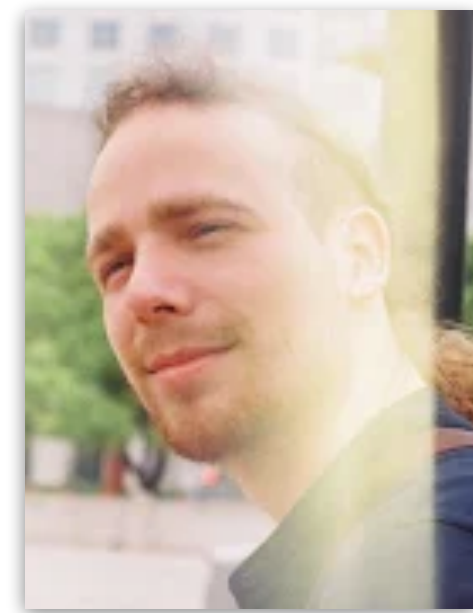


William S. Moses



wmoses@mit.edu
JuliaCon ESM MiniSymposium
July 25, 2022





William S. Moses Valentin Churavy Ludger Paehler Johannes Doerfert



Jan Hückelheim

Sri Hari Krishna
Narayanan

Michel Schanen

Paul Hovland

&
more



Leila Ghaffari

Praytush Das

Tim Gymnich

Manuel Drehwald

For more details see Tim's talk on Wednesday!!



Tim Gymnich

Fast Forward and Reverse-Mode Differentiation via Enzyme.jl

👥 [Valentin Churavy](#), [William Moses](#), [Ludger Paehler](#), [Tim Gymnich](#)

🕒 07/27/2022, 9:00 AM — 9:30 AM EDT

🏷️ Purple

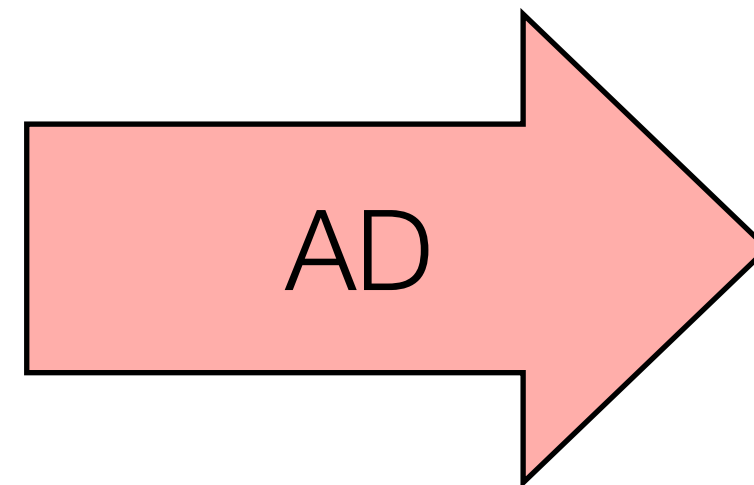
Abstract:

Enzyme is a new LLVM-based differentiation framework capable of creating fast derivatives in a variety of languages. In this talk we will showcase improvements in Enzyme.jl, the Julia-language bindings for Enzyme that enable us to differentiate through parallelism (Julia tasks, MPI.jl, etc), mutable memory, JIT-constructs, all while maintaining performance. Moreover we will also showcase Enzyme's new forward mode capabilities in addition to its existing reverse-mode features.

Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs

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



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

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

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

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

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

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


Existing AD Approaches (2/3)

- Operator overloading (ReverseDiff.jl, ForwardDiff.jl, Adept, JAX)
 - Differentiable versions of existing language constructs (Float64 => Dual{Float64})
 - May require writing to use non-standard utilities
 - Often dynamic: storing instructions/values to later be interpreted

```
function relu3(x::TrackedArray)
  if x[1] > 0
    return x[1]^3
  else
    return 0
  end
end
```

```
# Store all instructions into an
# instruction tape
gtape = GradientTape(relu3, ([2.0],))

# Interpret instructions on the tape
# to construct derivative
seeded_reverse_pass!(result, gtape)
```

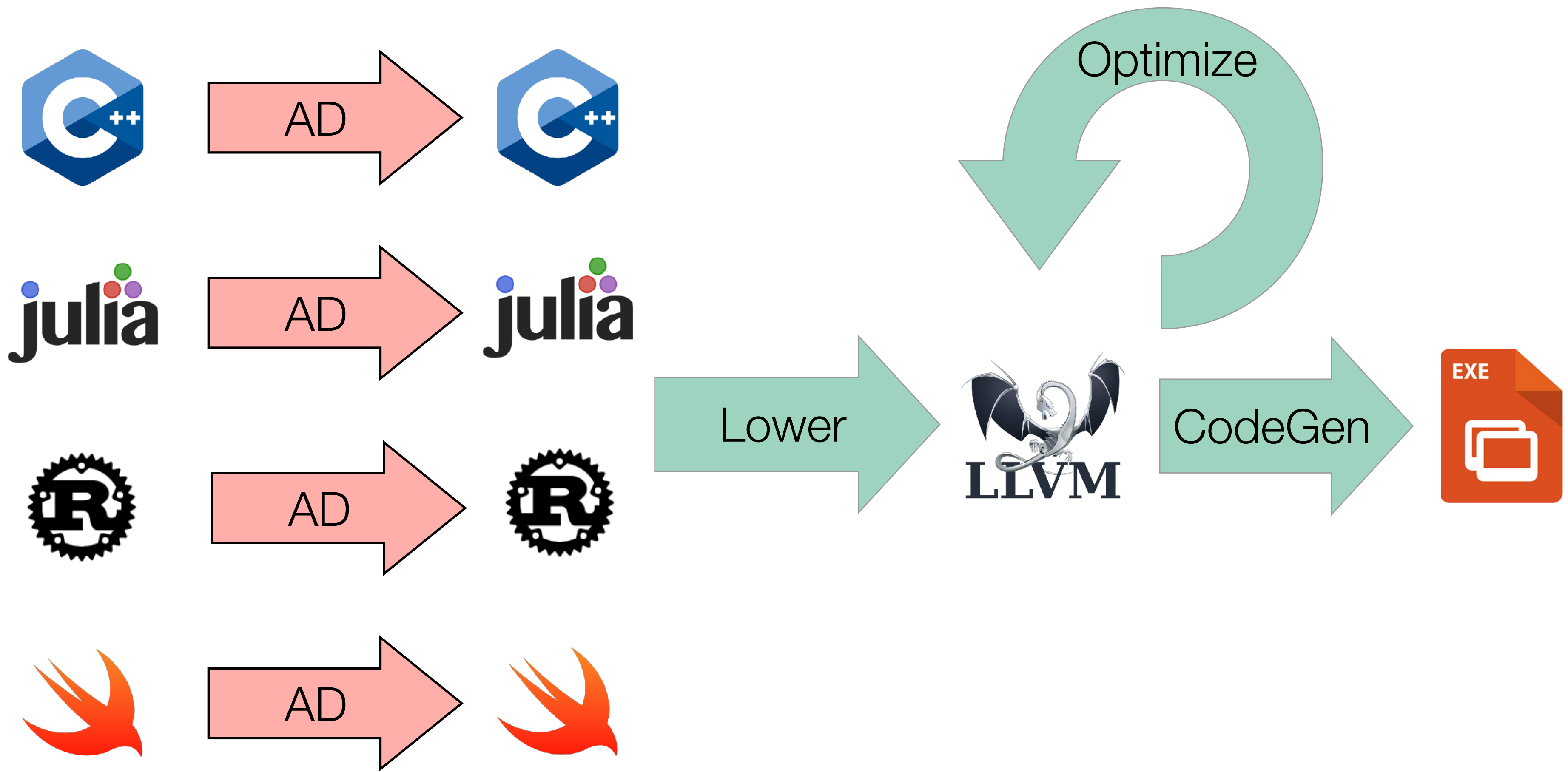


Existing AD Approaches (3/3)

- Source rewriting (Zygote.jl -ish, Tapenade)
 - 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)
function mag(x::Vector{Float64})::Float64

# Compute norm in O(n^2)
function norm(out::Vector{Float64},
              in::Vector{Float64})

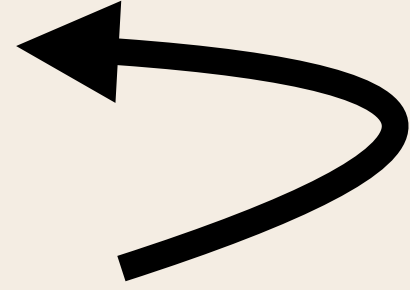
    for i = 1:n
        out[i] = in[i] / mag(in)
    end
end
```



Case Study: Vector Normalization

```
# Compute magnitude in O(n)
function mag(x::Vector{Float64})::Float64

# Compute norm in O(n^2)
function norm(out::Vector{Float64},
              in::Vector{Float64})
    res = mag(in)
    for i = 1:n
        out[i] = in[i] / res
    end
end
```



Optimization & Automatic Differentiation

$O(n^2)$

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

Optimize

$O(n)$

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

AD

$O(n)$

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

Optimization & Automatic Differentiation

$O(n^2)$

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

Optimize

$O(n)$

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

AD

$O(n)$

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

$O(n^2)$

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

AD

$O(n^2)$

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

Optimization & Automatic Differentiation

$O(n^2)$

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

Optimize

$O(n)$

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

AD

$O(n)$

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

$O(n^2)$

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

AD

$O(n^2)$

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

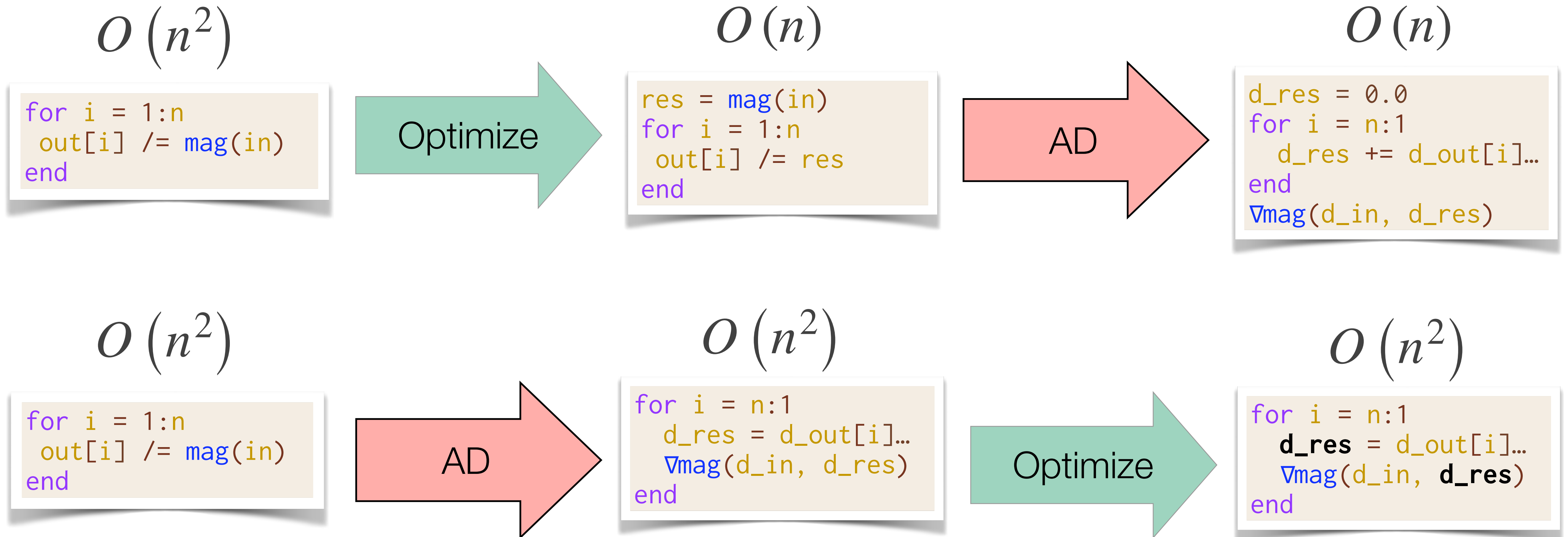
Optimize

$O(n^2)$

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

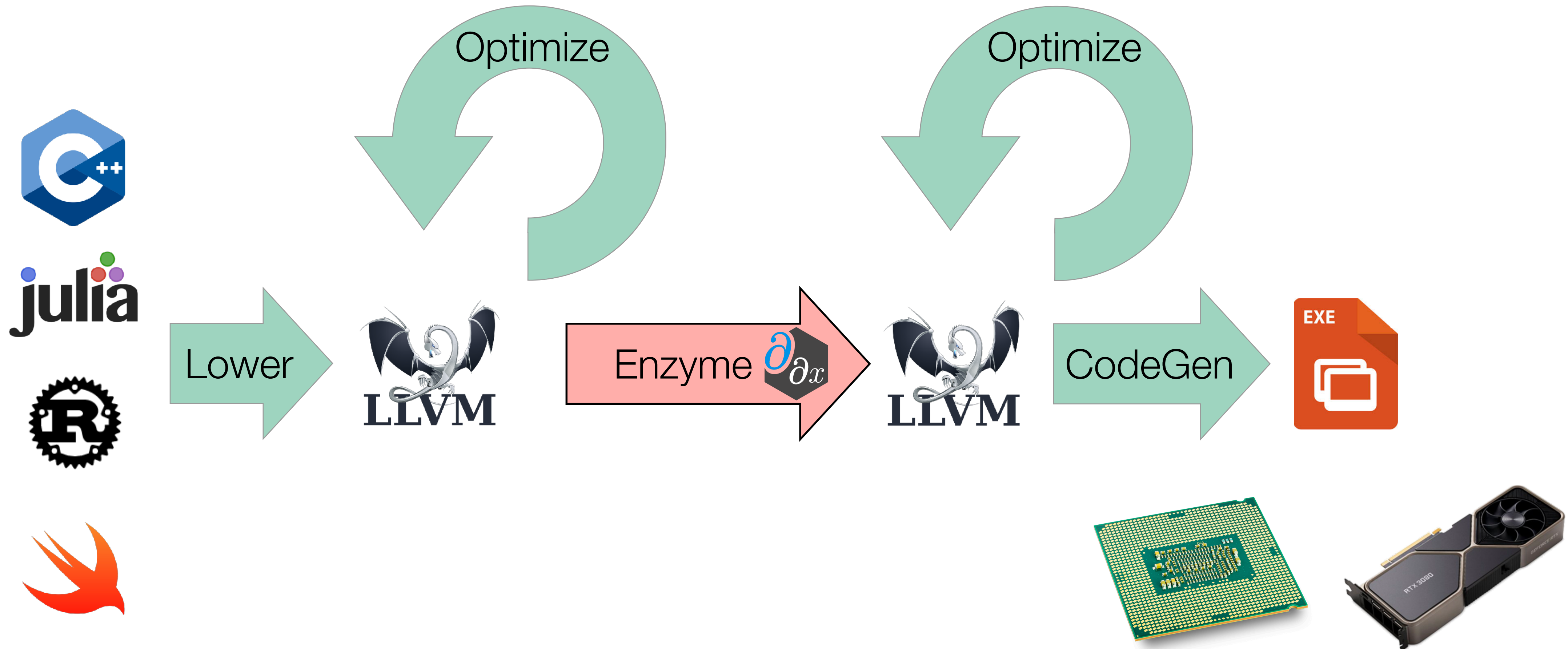
Optimization & Automatic Differentiation

Differentiating after optimization can create *asymptotically faster* gradients!

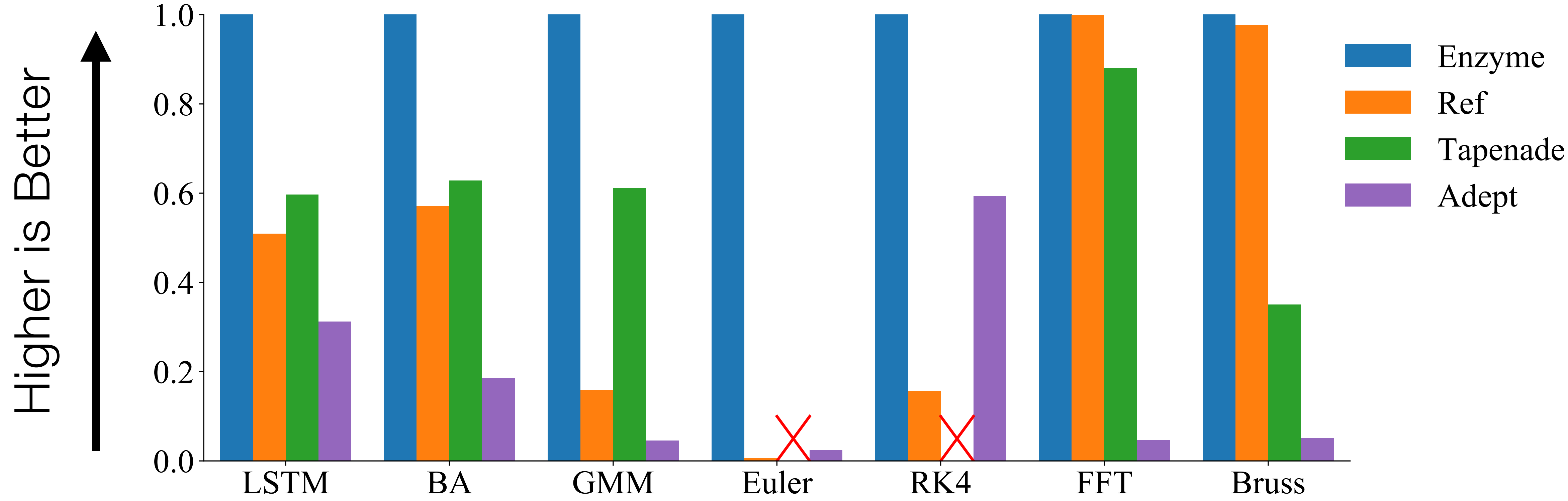


Enzyme Approach

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



Speedup of Enzyme [MC @ NeurIPS 2020]

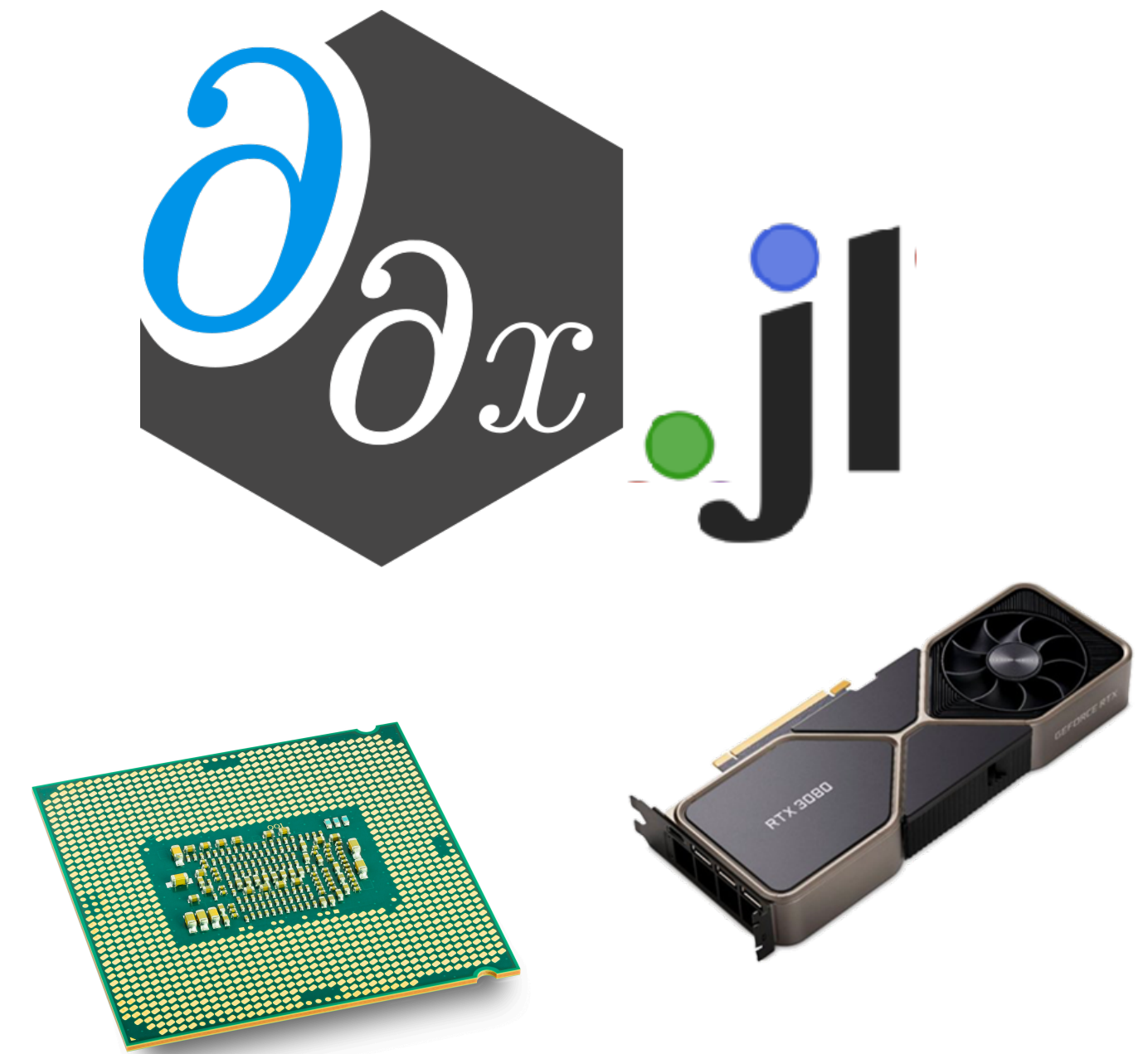


Enzyme is **4.2x faster** than Reference!



Enzyme.jl

- Julia bindings for Enzyme AD framework
 - Bindings built off of GPUCompiler.jl
- Forward and Reverse Mode AD, including experimental vector mode
- Handles mutation, parallelism, GPU's (AMD, CUDA, etc), & more!
- Static analysis & optimization => very, very fast scalar AD



- Static analysis & optimization => very, very fast scalar AD

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

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

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

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

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

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

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

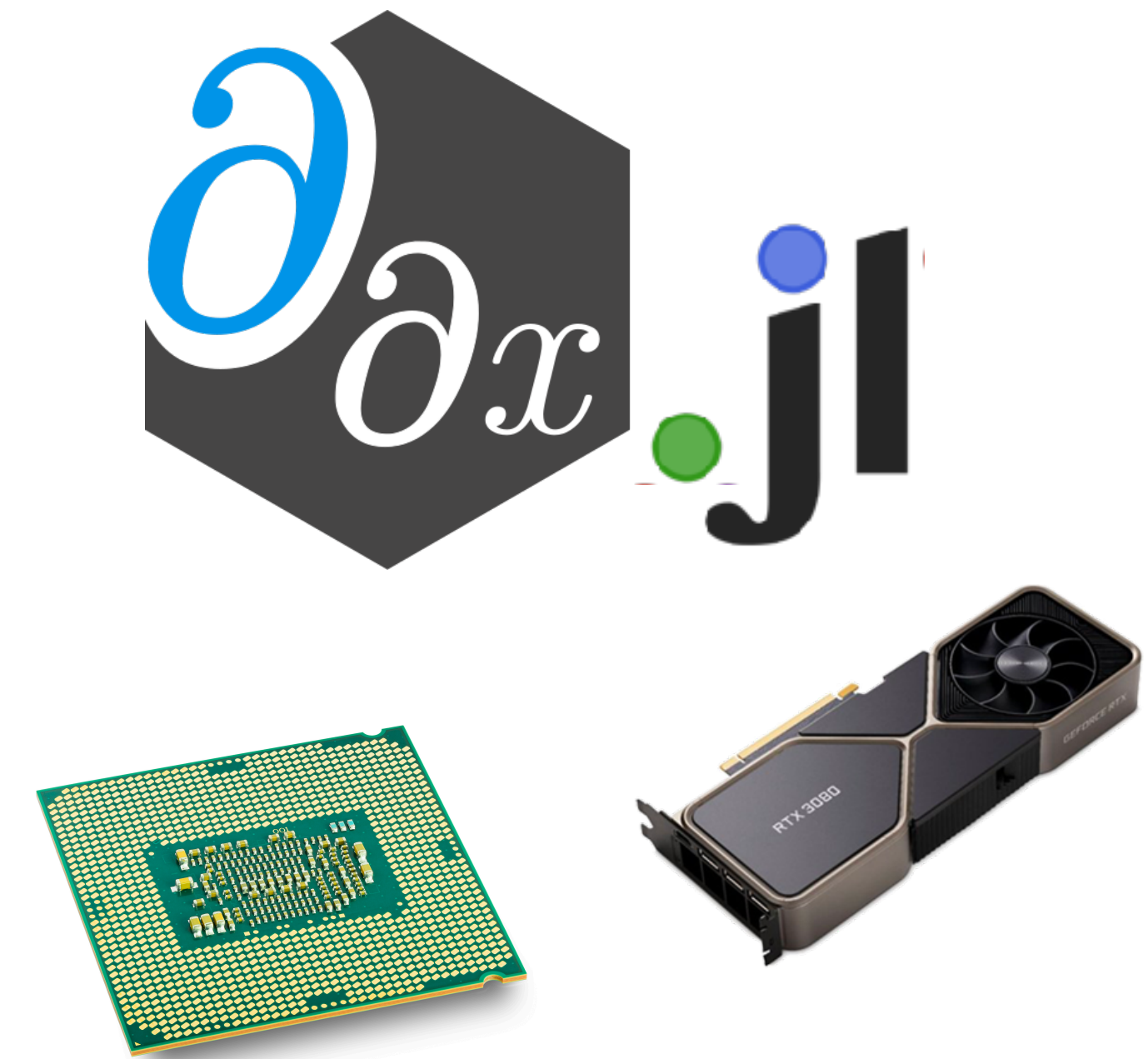
@btime Diffraction.gradient(taylor, 0.5, 10^6)
# 96665 ms (96.37 GiB)

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

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

Enzyme.jl, the Sharp Bits

- Built off of GPUCompiler.jl
 - GPU-style code is supported, generic code is in progress
 - Type Stability!
- Only BLAS, not LAPACK currently supported
- No ChainRules Support (precursor EnzymeRules coming soon)
- Internal allocations may hit in progress GC-support



Automatic Differentiation & GPUs [MCPHNSD @ SC'21]

- 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



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)

```
function set(ar::Vector{Float64},  
            val::Ref{Float64})  
    @threads for i=1:10  
        ar[i] = val[]  
    end  
end
```

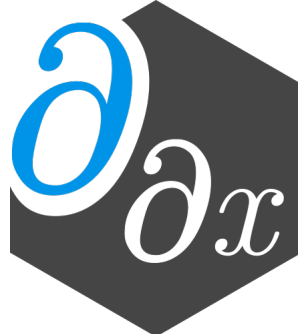
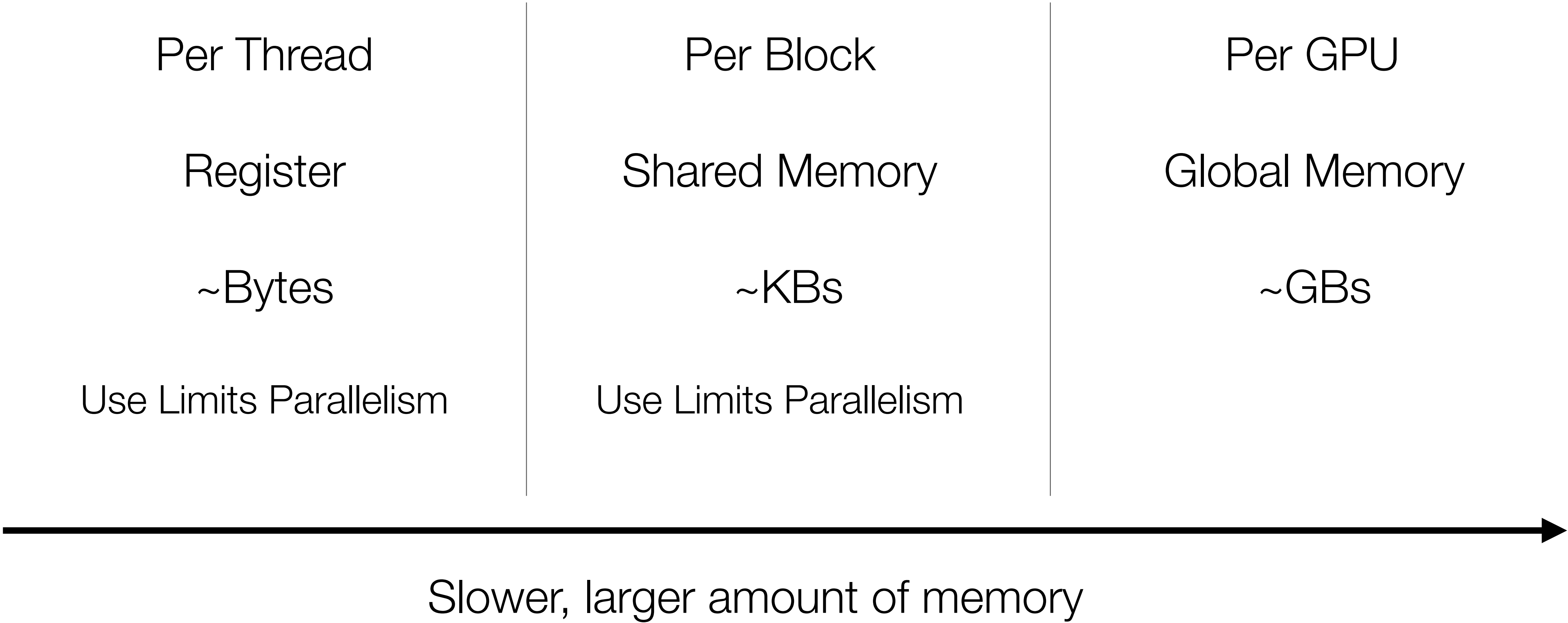
Read Race

```
function grad_set(ar::Vector{Float64},  
                 grad_ar::Vector{Float64},  
                 val::Ref{Float64})  
    d_val = Ref(0.0)  
  
    @threads for i=1:10  
        ar[i] = val[]  
    end  
  
    @threads for i=1:10  
        d_val[] += d_ar[i]  
        d_ar[i] = 0.0  
    end  
  
    return d_val[]  
end
```

Write Race



GPU Memory Hierarchy



Correct and Efficient Derivative Accumulation

Thread-local memory

- Non-atomic load/store

```
# Device function
function f(...)

  # Thread-local var
  y::Float64

  ...

  d_y += val;
end
```

Same memory location across all threads (some shared mem)

- Parallel Reduction

```
// Same var for all threads
const y::Ref{Float64}

# Device function
function f(...)

  ...

  reduce_add(d_y, val);
end
```

Others [always legal fallback]

- Atomic increment

```
# Device function with
# Unknown thread-aliasing
function f(y::Vector{Float64})
  ...

  atomic
    d_y[...] += val;
  end
end
```

Slower

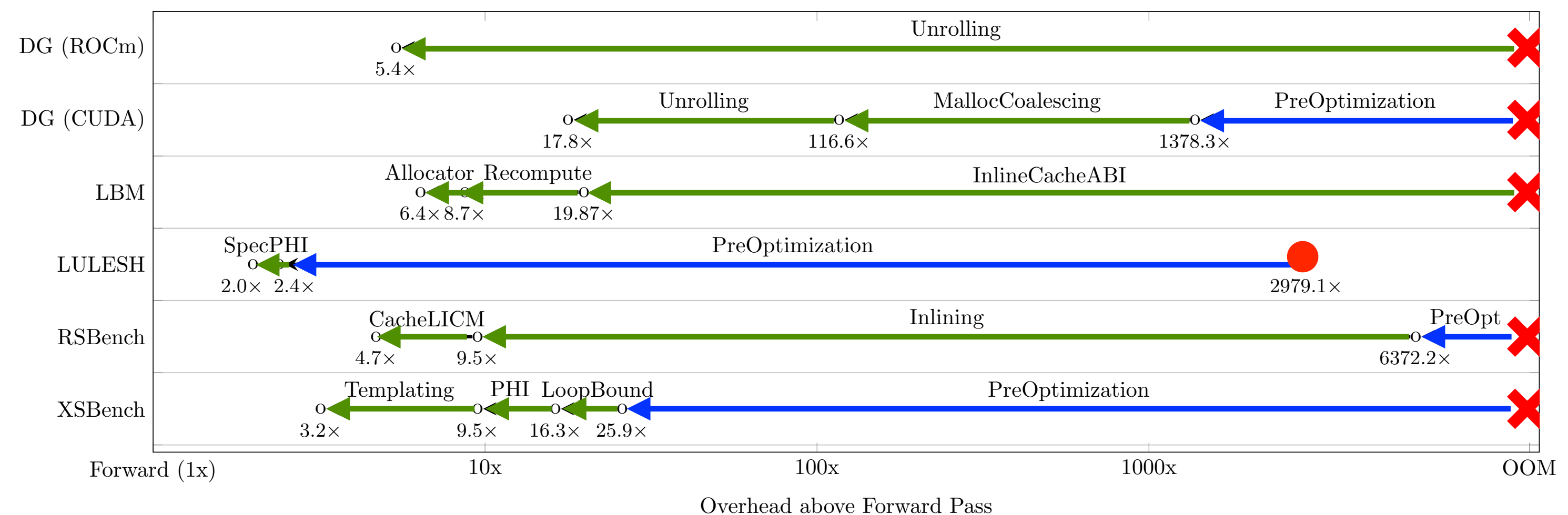


Novel AD + GPU Optimizations

- Reduce runtime by up to 3 orders of magnitude & not OOM!
- See our paper for full list (<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
- [GPU] Merge Allocations
- [GPU] Aliasing of SyncThreads

• ...



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))
```

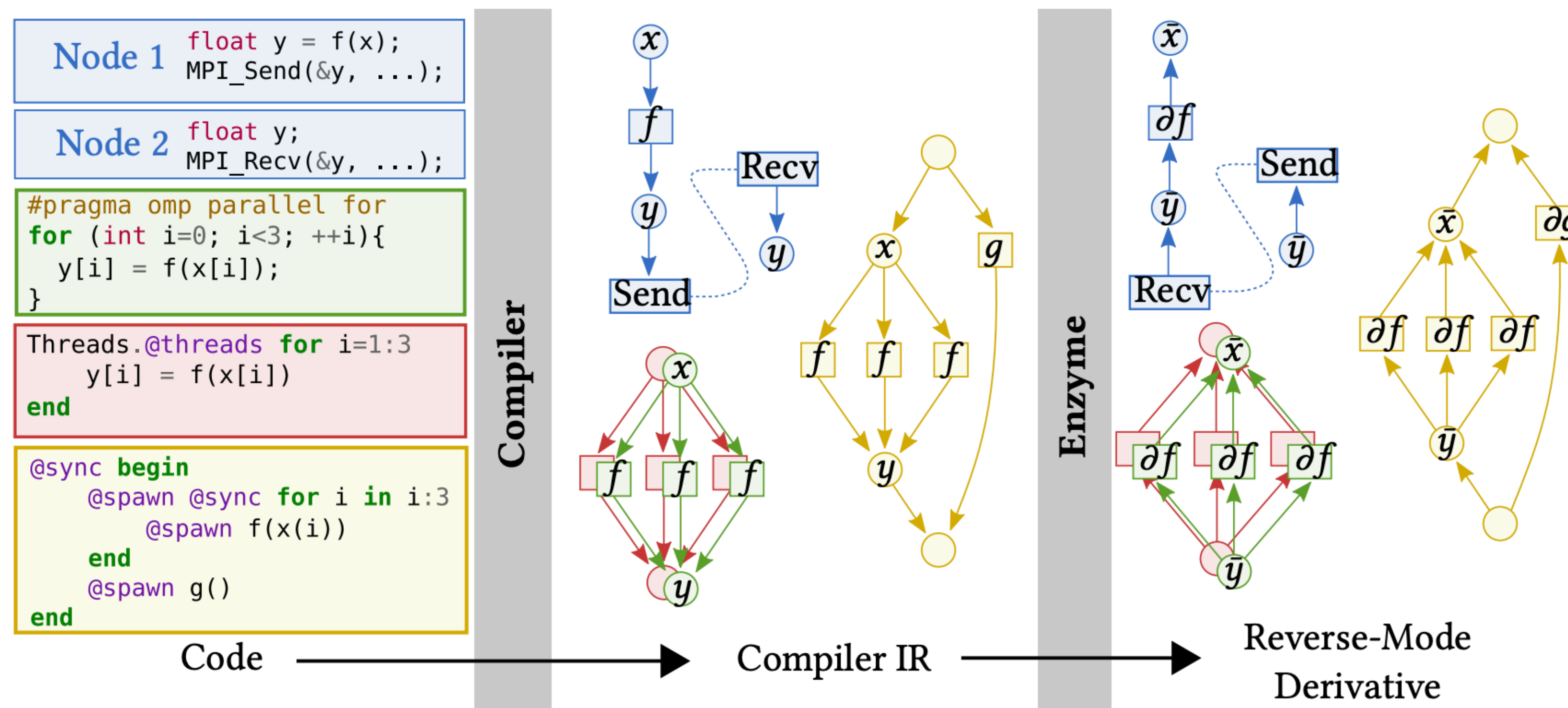
See Below For Full Code Examples

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



Common Framework for Parallel AD [To Appear at SC'22]

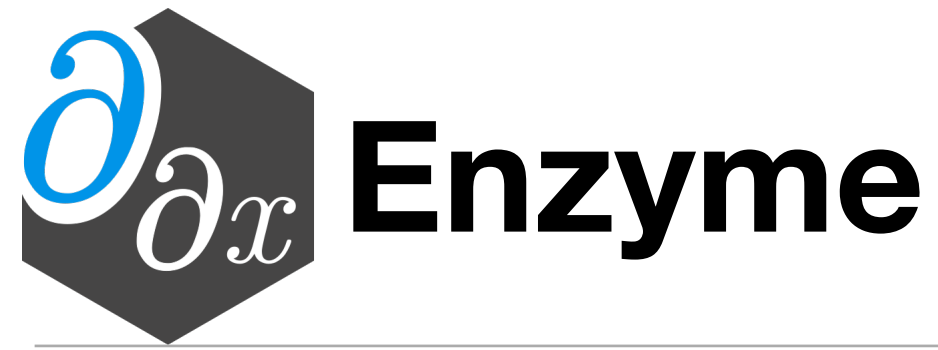
- Common infrastructure for supporting parallel AD (caching, race-resolution, gradient accumulation) enables parallel differentiation independent of framework or language.



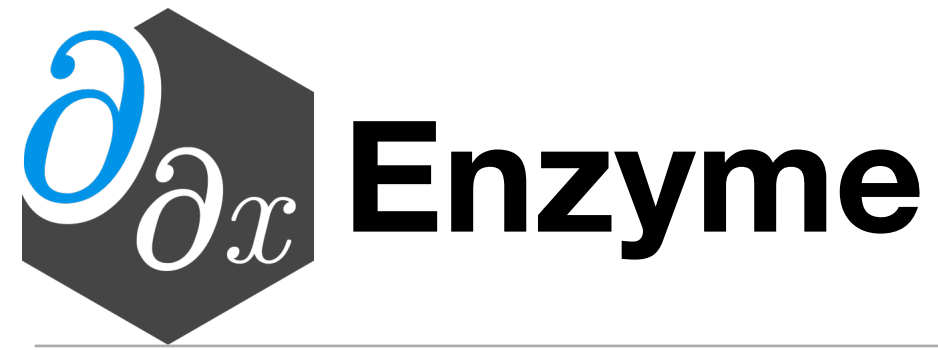
- Enables differentiation of a combination of GPU (e.g. CUDA, ROCm), CPU (OpenMP, Julia Tasks, RAJA), Distributed (MPI, MPI.jl), and more







- Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)
- 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 & join our mailing list)!

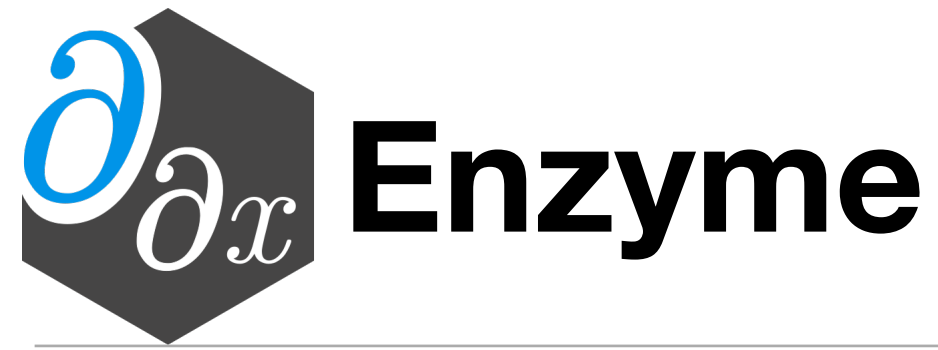


- Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)
- 4.2x speedup over AD before optimization on CPU
- State-of-the 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 & join our mailing list)!

Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Lilly Chin, Hal Finkel, Marco Foco, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Vassil Vassilev, and Alex Zinenko
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323. Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443. Ludger Paehler was supported in part by the German Research Council (DFG) under grant agreement No. 326472365.
- 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.
- 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.





- Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)
- 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 & join our mailing list)!

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);
}
```



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];
...

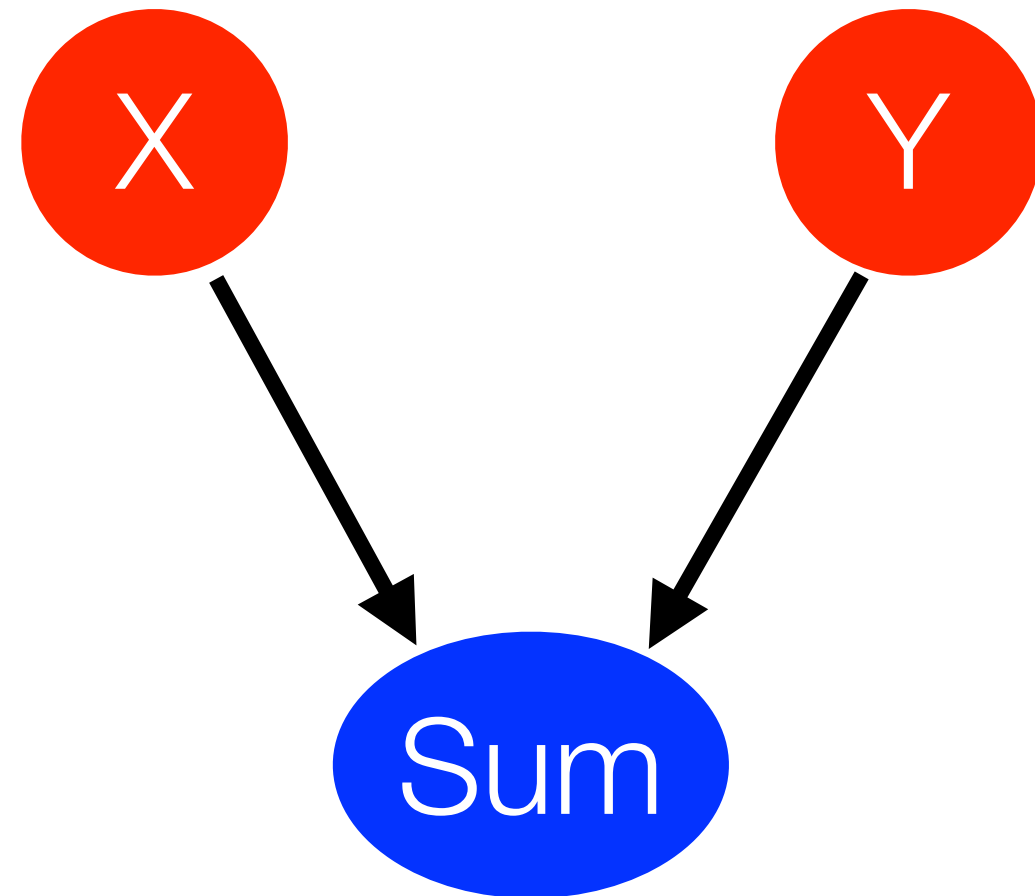
delete[] x_cache;
```



Cache Reduction Example

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

Overwritten:



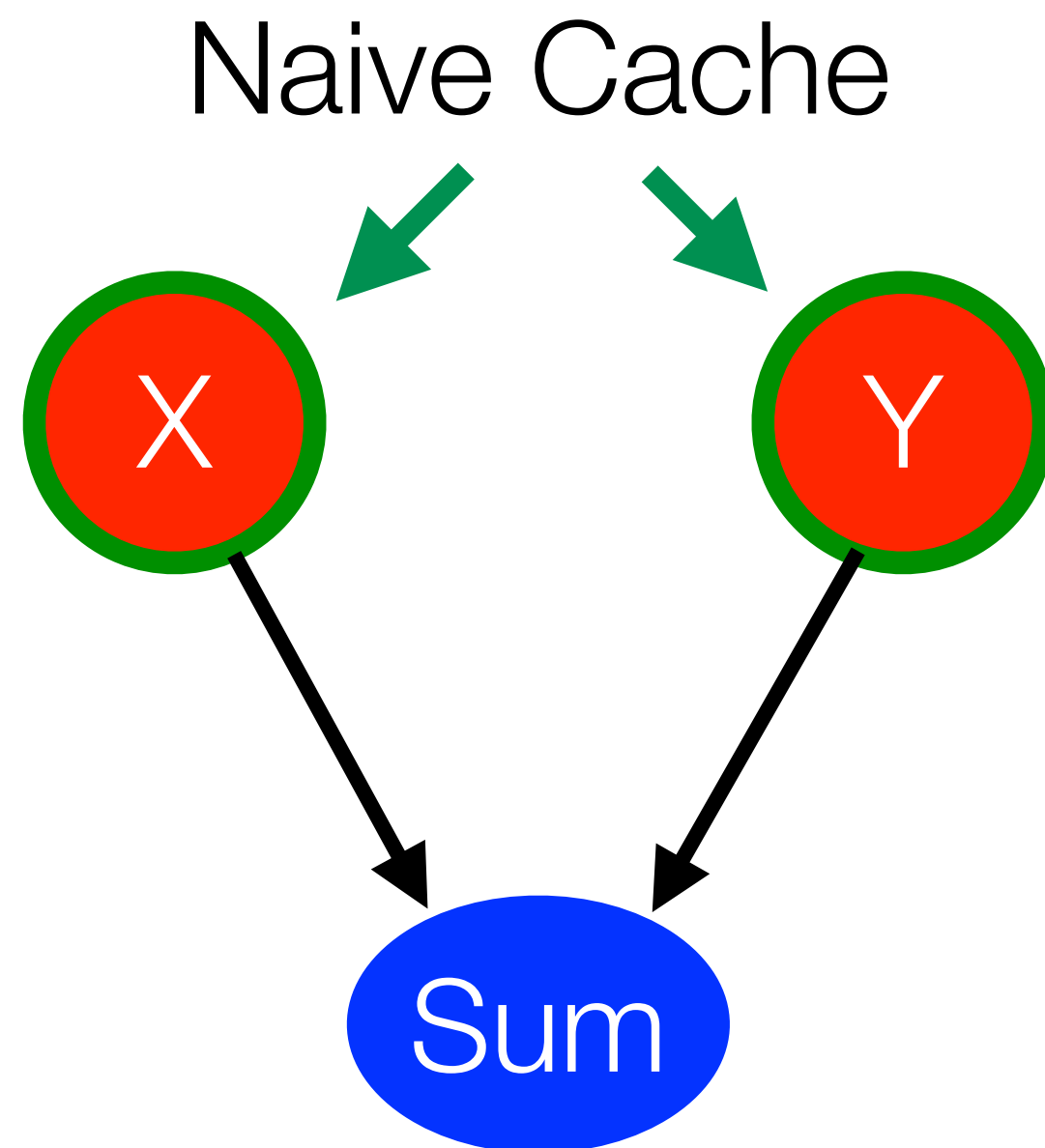
Required for Reverse:

```
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.

Overwritten:



Required for Reverse:

```
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];

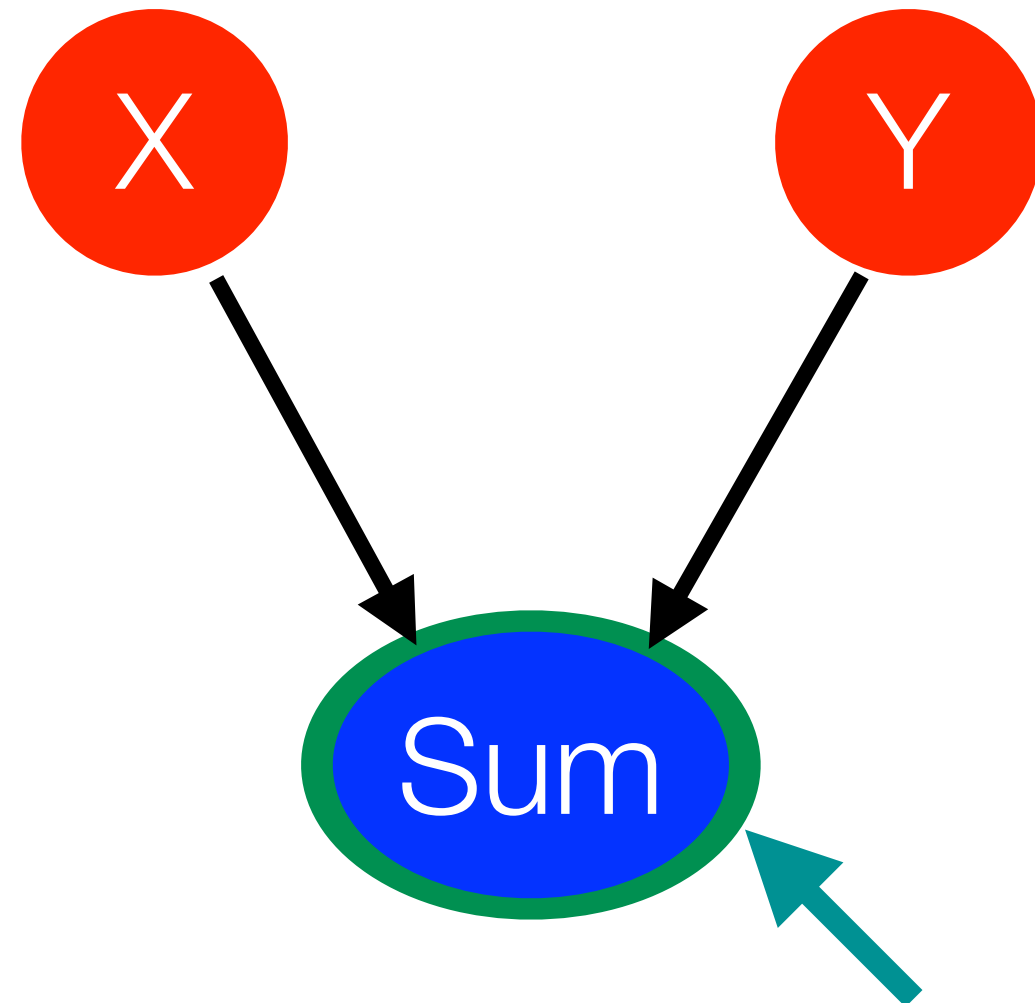
for(int i=0; i<10; i++) {
    double sum = x[i] + y[i];
    sum_cache[i] = sum;

    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

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;
```



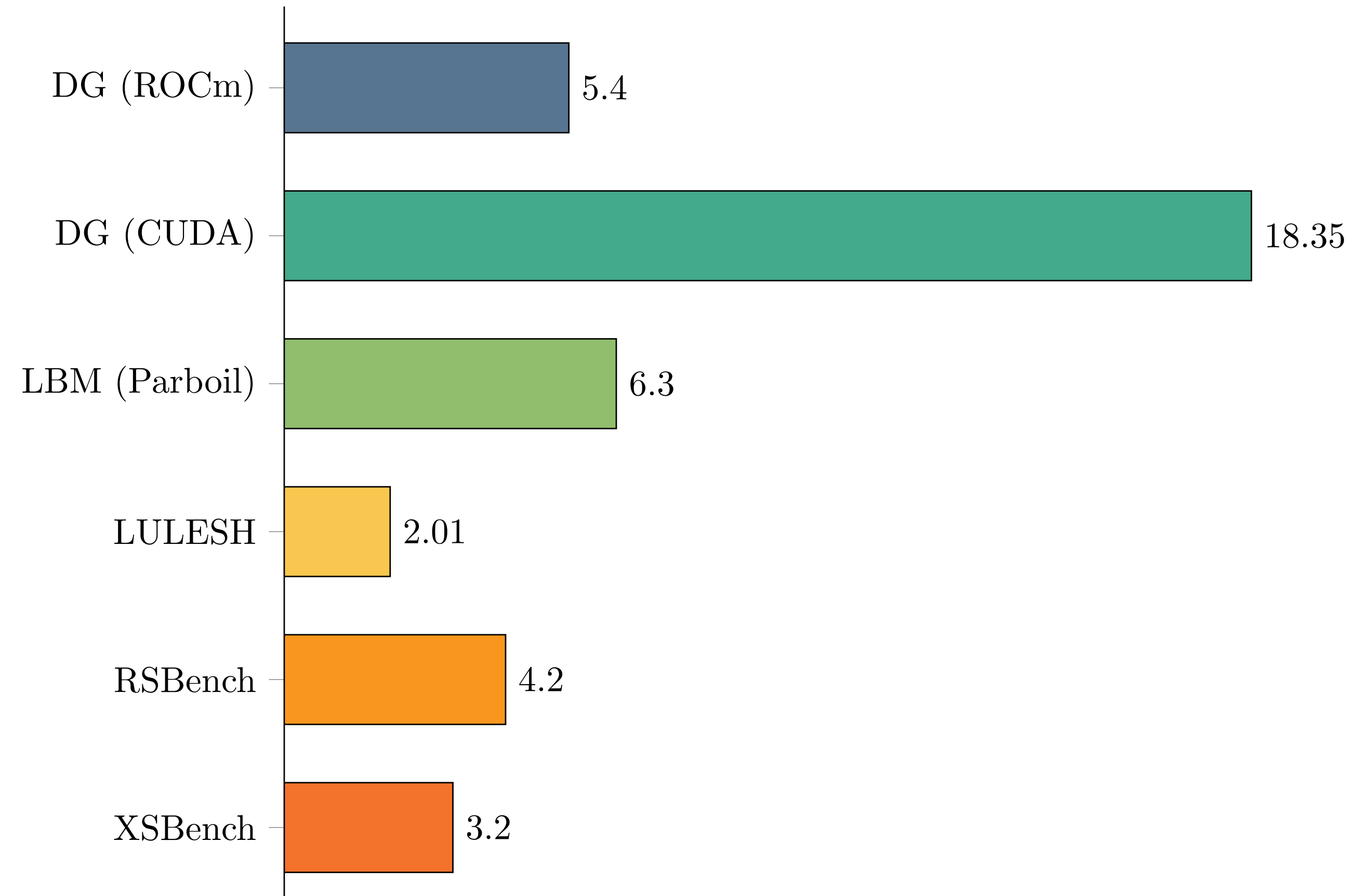
Novel AD + GPU Optimizations

- See our SC paper (Nov 17) 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
- ...



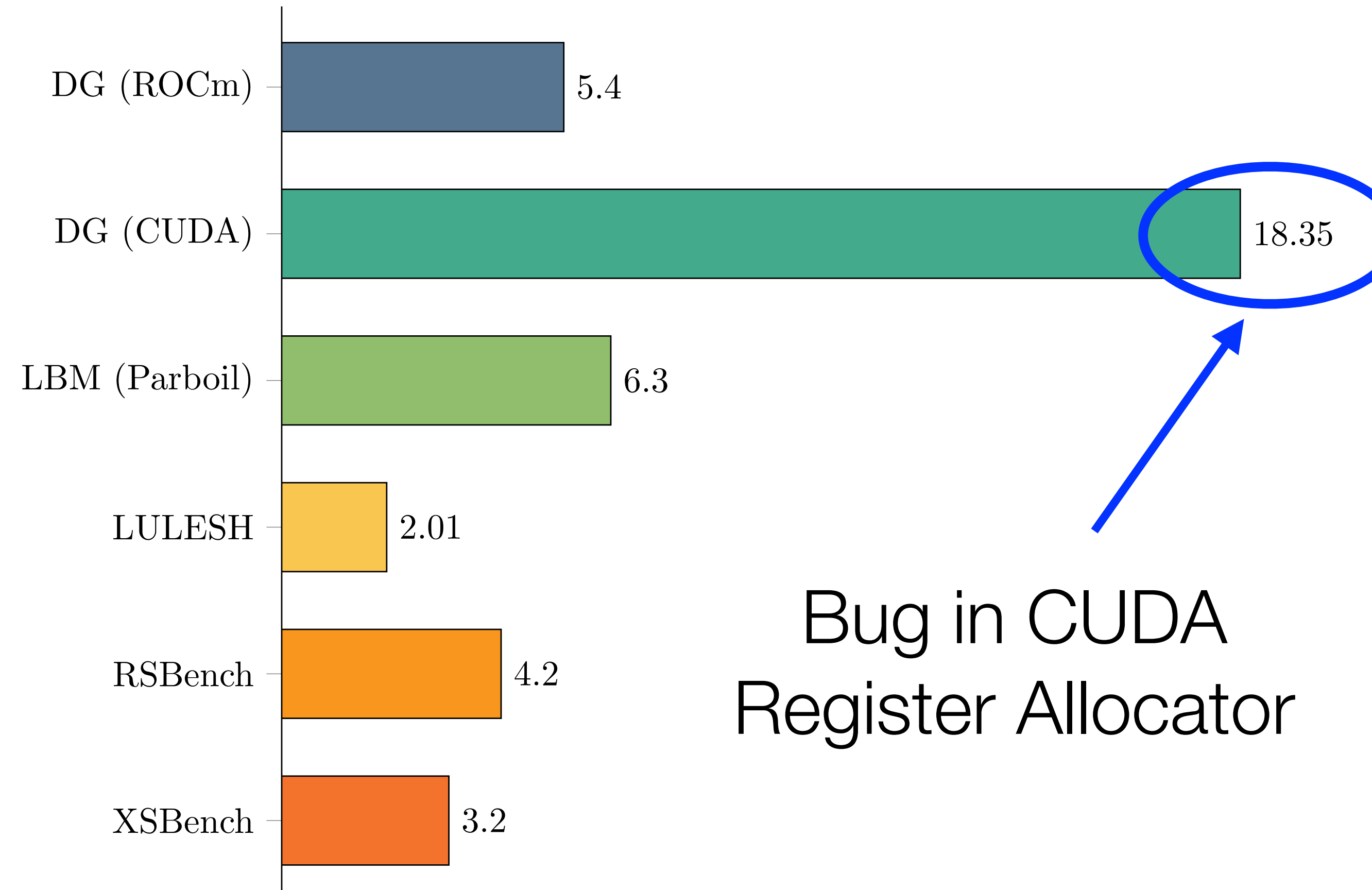
GPU Gradient Overhead [MCPHNMJ'21]

- 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 [MCPHNMJ'21]

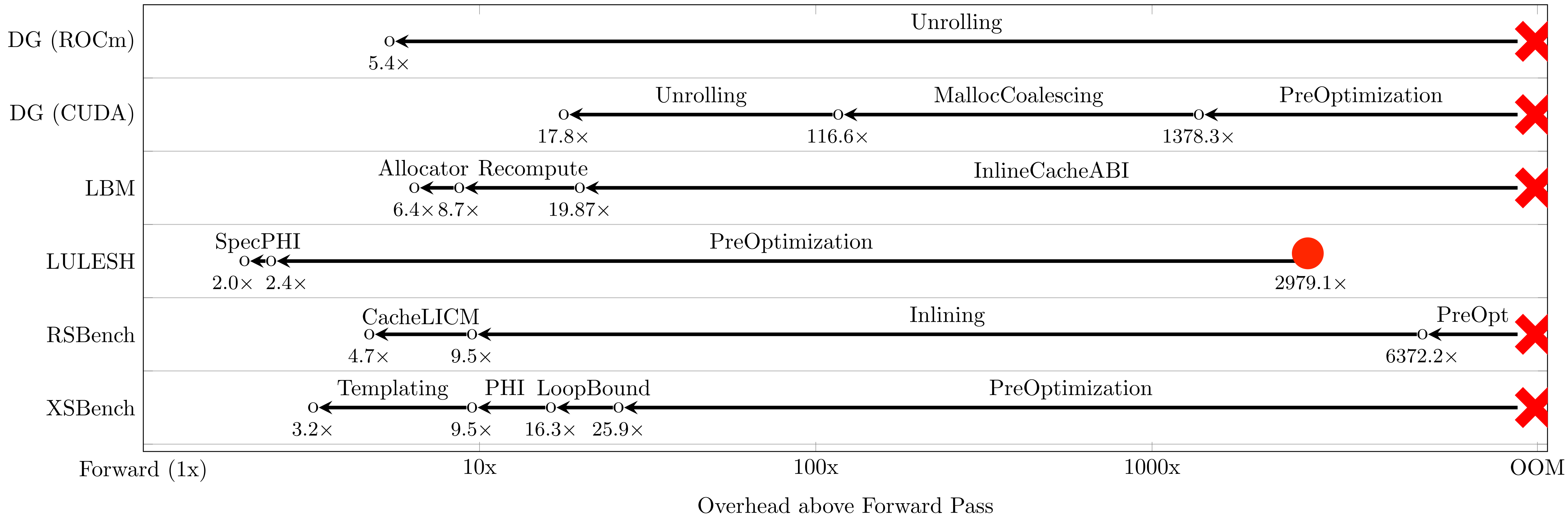
- Evaluation of both original code and gradient
 - DG: Discontinuous-Galerkin integral (Julia)
 - LBM: particle-based fluid dynamics simulation
 - LULESH: unstructured explicit shock hydrodynamics solver
 - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute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



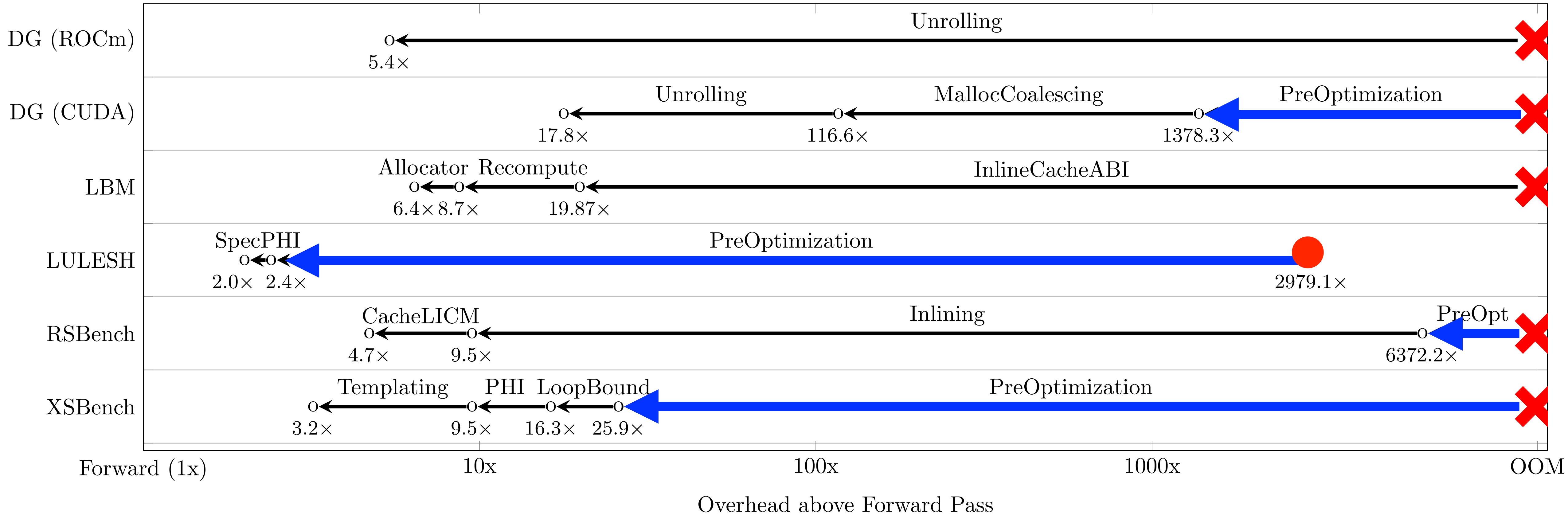
Bug in CUDA
Register Allocator



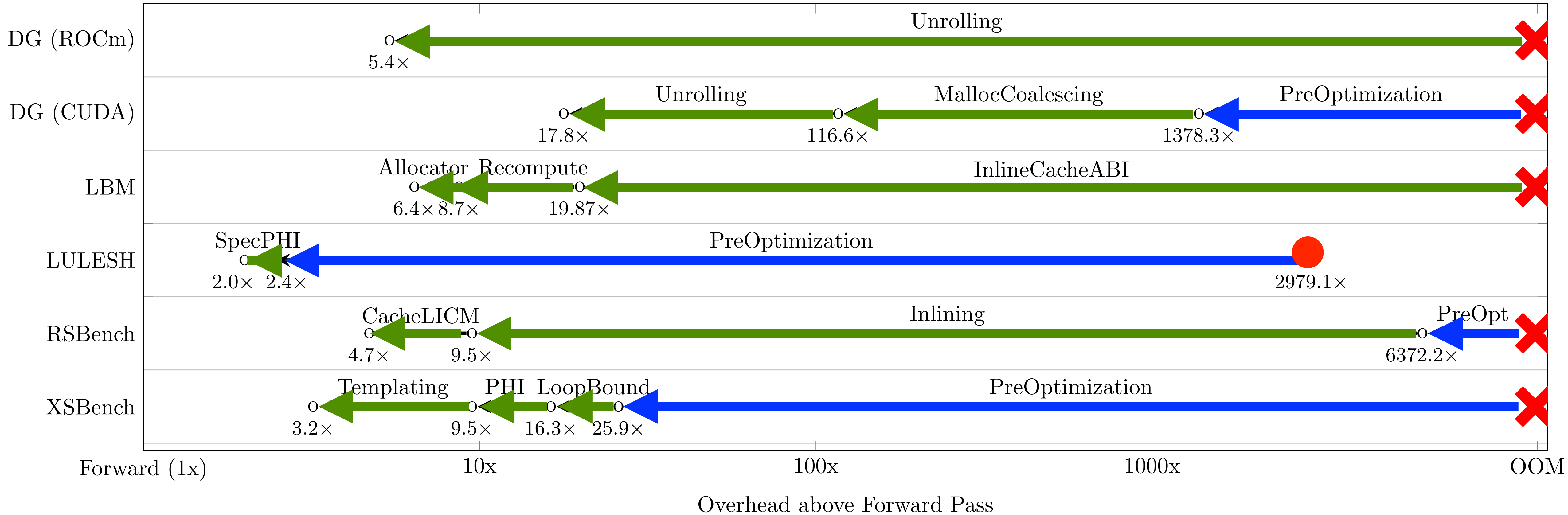
Ablation Analysis of Optimizations [MCPHNMJ'21]



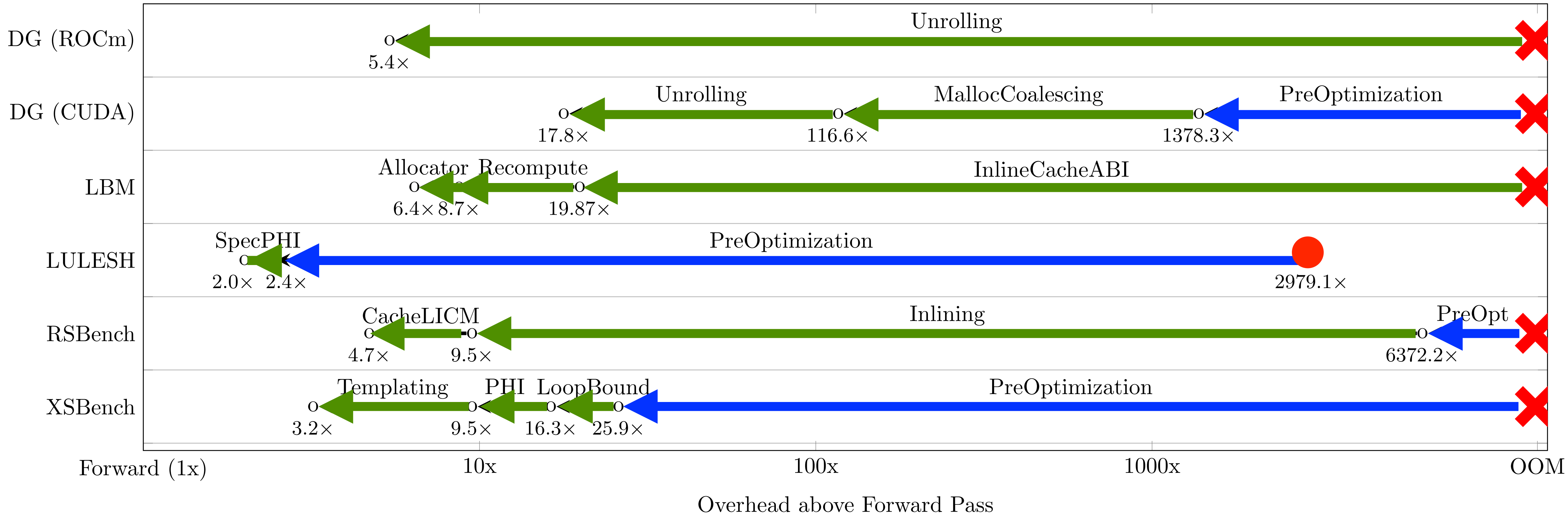
Ablation Analysis of Optimizations [MCPHNMJ'21]



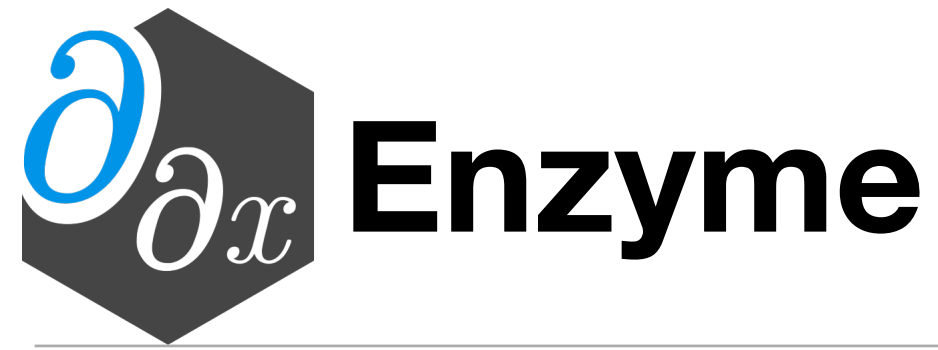
Ablation Analysis of Optimizations [MCPHNMJ'21]



Ablation Analysis of Optimizations [MCPHNMJ'21]



GPU AD is Intractable Without Optimization!

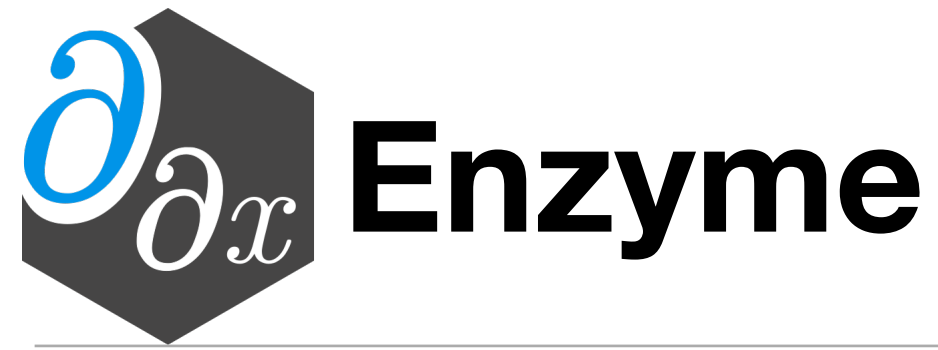


- Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)
- 4.2x speedup over AD before optimization on CPU
- State-of-the 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 & join our mailing list)!

Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Lilly Chin, Hal Finkel, Marco Foco, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Vassil Vassilev, and Alex Zinenko
- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323. Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443. Ludger Paehler was supported in part by the German Research Council (DFG) under grant agreement No. 326472365.
- 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.
- 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.





- Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)
- 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 & join our mailing list)!

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; j<M; j++) {
        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
```

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
```

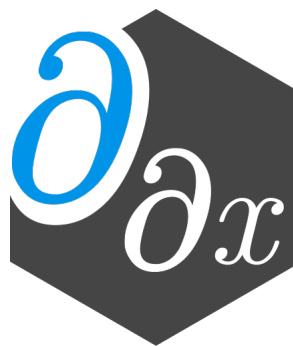
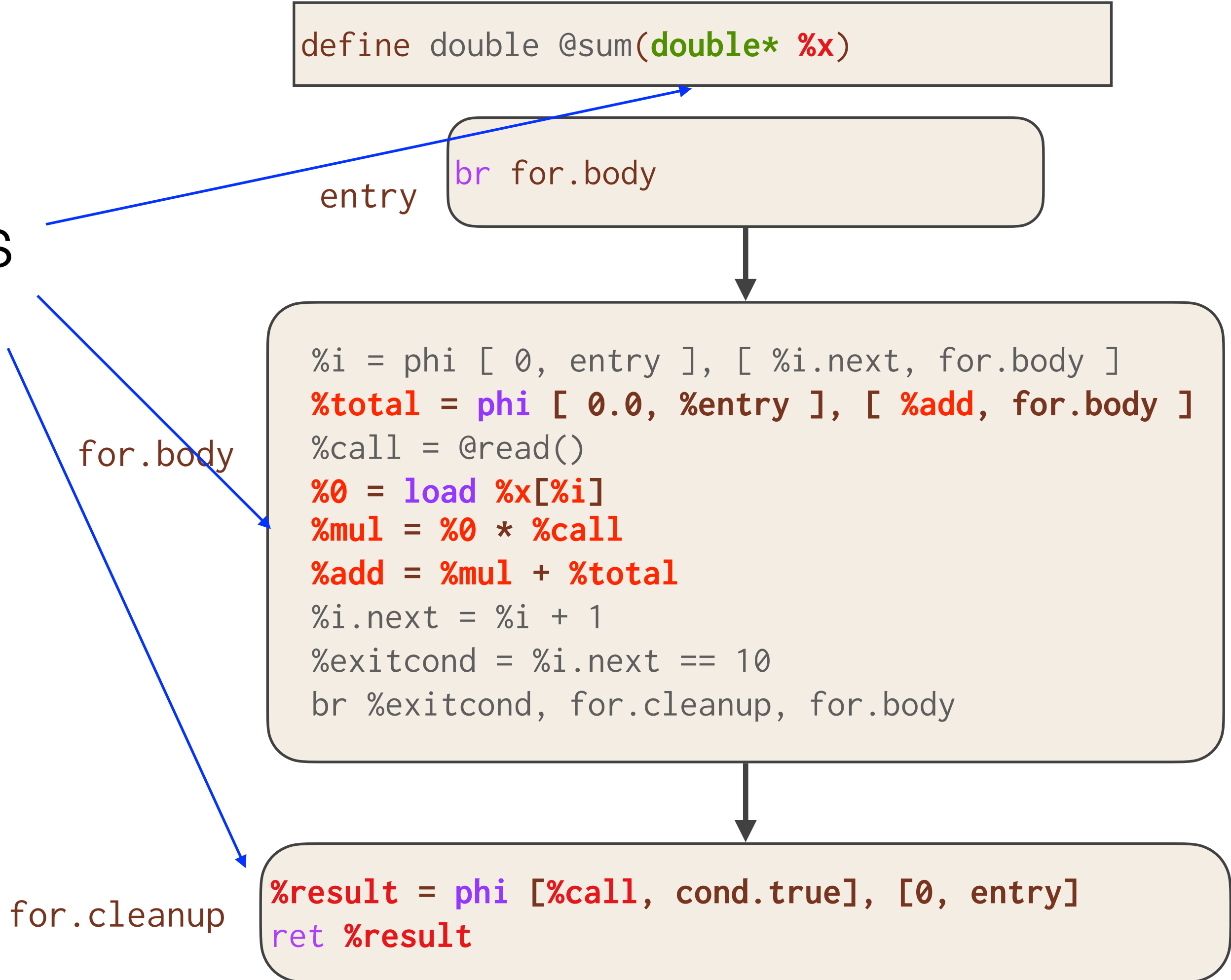
for.cleanup

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



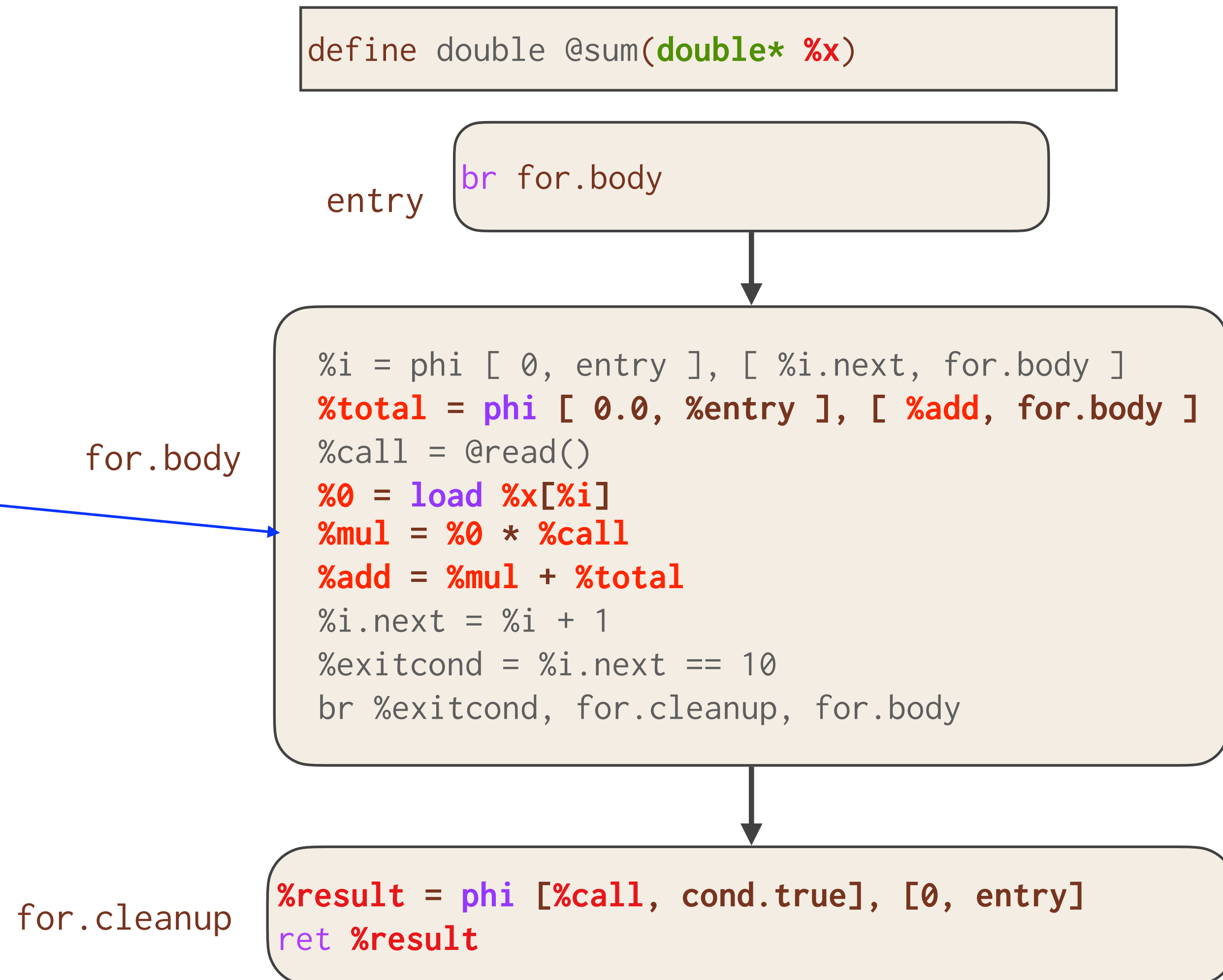
Case Study: Read Sum

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();
}
```



Enzyme on the GPU

- Care must be taken to both ensure correctness and maintain parallelism.
- GPU programs have much lower memory limits. Performance is highly dependent on the number of memory transfers.
- Without first running optimizations reverse-mode AD of large kernels is intractable (OOM).
- Novel GPU and AD-specific optimizations can make a difference of several orders of magnitude when computing gradients.

Test	Overhead
Forward	1
AD, Optimized	4.4
AD, No CacheLICM	343.7
AD, Bad Recompute Heuristic	1275.6
AD, No Inlining	6372.2
AD, No PreOptimization	OOM



CUDA Automatic Differentiation

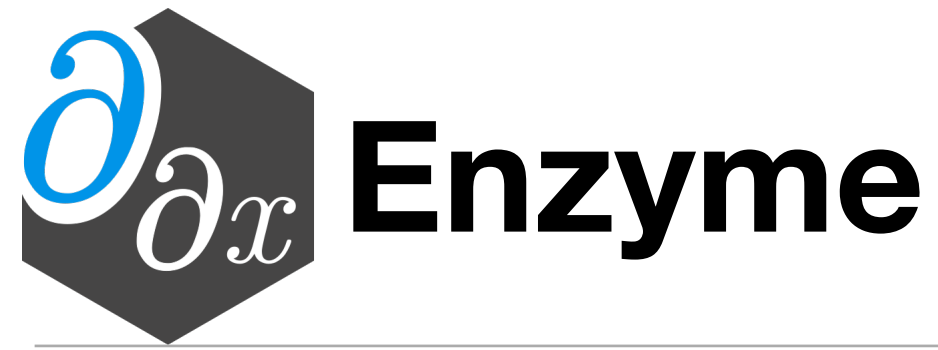
- Enzyme enables differentiation of CPU programs without rewriting them in a DSL.
- Similarly, GPU programs cannot currently be differentiated without being rewritten in a differentiable language (e.g. PyTorch).
- Enzyme enables reverse-mode AD of general existing GPU programs by:
 - Resolving potential 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





- Tool for performing 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
- State-of-the art performance with existing tools
- Differentiate GPU kernels
- Open Source (enzyme.mit.edu / github.com/wsmoses/Enzyme)
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow

GPU Automatic Differentiation

- Prior work has not explored reverse mode AD of GPU kernels
- 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





- Tool for performing 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
- State-of-the art performance with existing tools
- Differentiate GPU kernels
- Open Source (enzyme.mit.edu / github.com/wsmoses/Enzyme)
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow

Custom Derivatives & Multisource

- One can specify custom forward/reverse passes of functions by attaching metadata

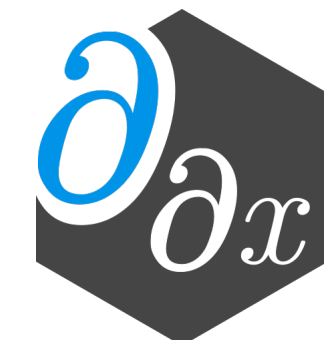
```
__attribute__((enzyme("augment", augment_func)))  
__attribute__((enzyme("gradient", gradient_func)))  
double func(double n);
```

- Enzyme leverages LLVM's link-time optimization (LTO) & "fat libraries" to ensure that LLVM bitcode is available for all potential differentiated functions before AD



CUDA Performance Improvements

- Introduce optimizations to reduce the use of memory
 - Alias Analysis to determine legality of recomputing an instruction
 - More aggressive alias analysis properties of syncthreads
 - Don't cache unnecessary values
 - Move cache outside of loops when possible
 - Heap-to-stack [and to register]
 - Don't cache memory itself acting as a cache [such as shared memory]



Enzyme Differentiation Algorithm

- Type Analysis
- Activity Analysis
- Synthesize derivatives
 - Forward pass that mirrors original code
 - Reverse pass inverts instructions in forward pass (adjoints) to compute derivatives
- Optimize



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

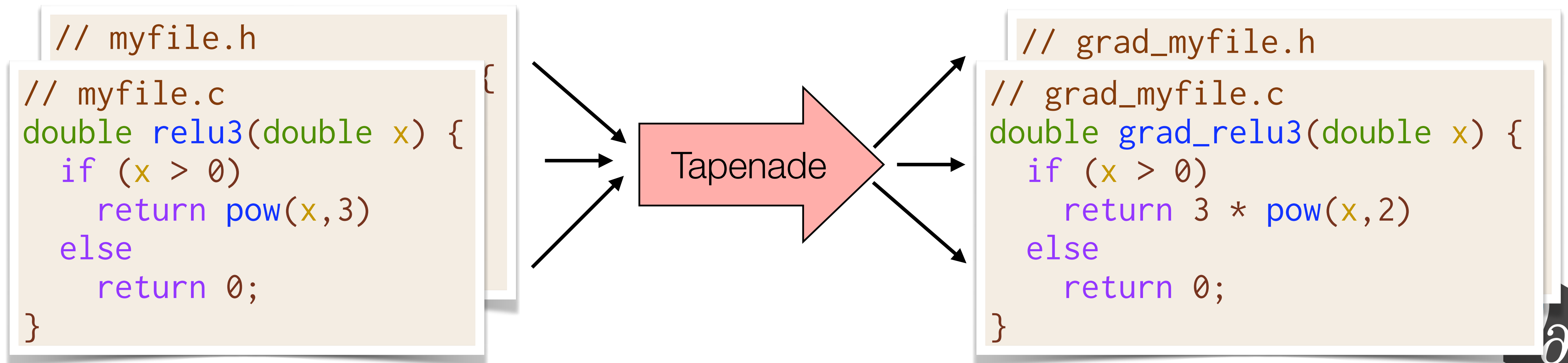
- Automatic Differentiation requires high level semantics to produce gradients
- Lack of high-level information can hinder performance of low-level AD
 - “AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR [...]”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

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



Existing AD Approaches (3/3)

- Source rewriting (Zygote.jl -ish, Tapenade)
 - 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



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
```

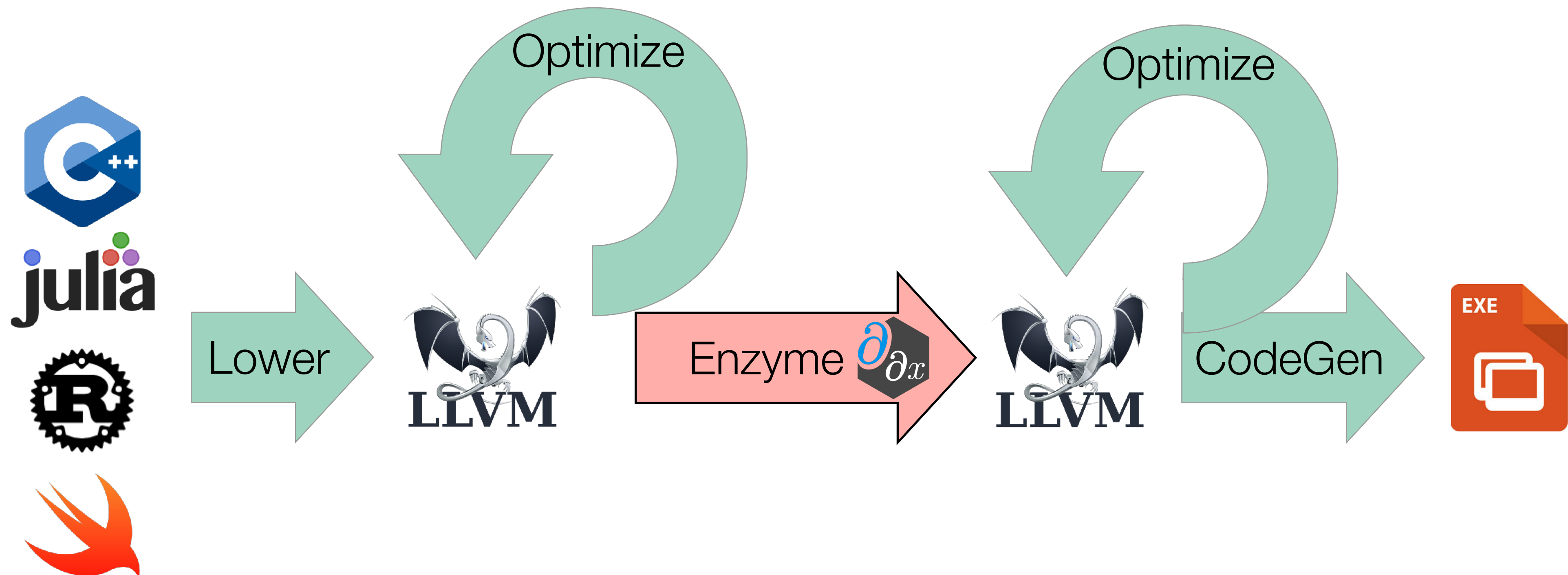
- Hinders application of ML to new domains
- Synthesizing gradients aims to close this gap





Enzyme Overturns Conventional Wisdom

- As fast or faster than state-of-the-art tools
 - Running after optimization enables a **4.2x speedup**
- Necessary semantics for AD derived at low-level (with potential cooperation of frontend)



Parallel Memory Detection

- Thread-local memory
 - Non-atomic load/store
- Same memory location across all threads
 - Parallel Reduction
- Others [always legal fallback]
 - Atomic increment

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



AD-Specific Cache

- Some optimizations require domain-specific knowledge
- Not all values are needed for the reverse pass. By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.
- Not all (loop) sizes are known at compile-time, so this must be a heuristic

```
double xy_cache=x[0] + y[0];  
  
use(x[0] + y[0]);  
  
overwrite(x, y);  
grad_overwrite(x, y);  
  
grad_use(xy_cache);
```

AD-Specific Cache

- Some optimizations require domain-specific knowledge
- Not all values are needed for the reverse pass. By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.
- Not all (loop) sizes are known at compile-time, so this must be a heuristic

```
double x_cache=x[0];  
double y_cache=y[0];  
  
use(x[0] + y[0]);  
  
overwrite(x, y);  
grad_overwrite(x, y);  
  
grad_use(x_cache + y_cache);
```



AD-Specific Cache

- Some optimizations require domain-specific knowledge
- Not all values are needed for the reverse pass. By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.
- Not all (loop) sizes are known at compile-time, so this must be a heuristic

```
double xy_cache=x[0] + y[0];  
  
use(x[0] + y[0]);  
  
overwrite(x, y);  
grad_overwrite(x, y);  
  
grad_use(xy_cache);
```

Differentiation Is Key To Machine Learning And Science

- Computing derivatives is key to many algorithms
 - Machine learning (back-propagation, Bayesian inference, uncertainty quantification)
 - Scientific computing (modeling, simulation)
- When working with large codebases or dynamically-generated programs, manually writing derivative functions becomes intractable
- Community has developed tools to create derivatives automatically



Existing AD Approaches

- 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
- Operator overloading (Adept, JAX)
 - Provide 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



Existing AD Approaches

- 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
 - Difficult to use with external libraries



Case Study: ReLU3

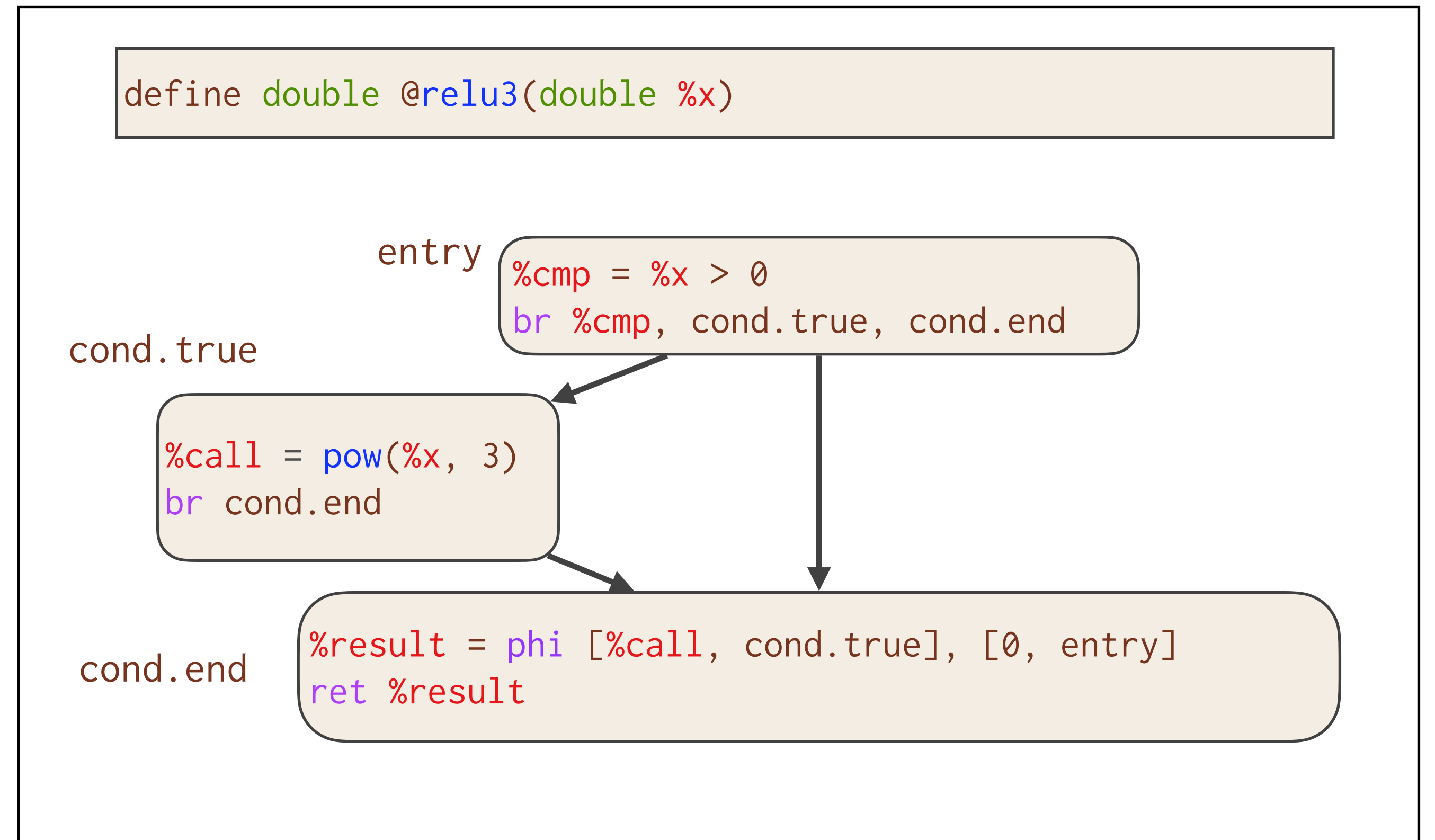
C Source

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

Enzyme Usage

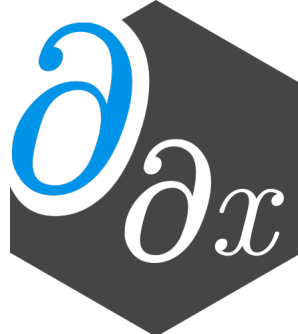
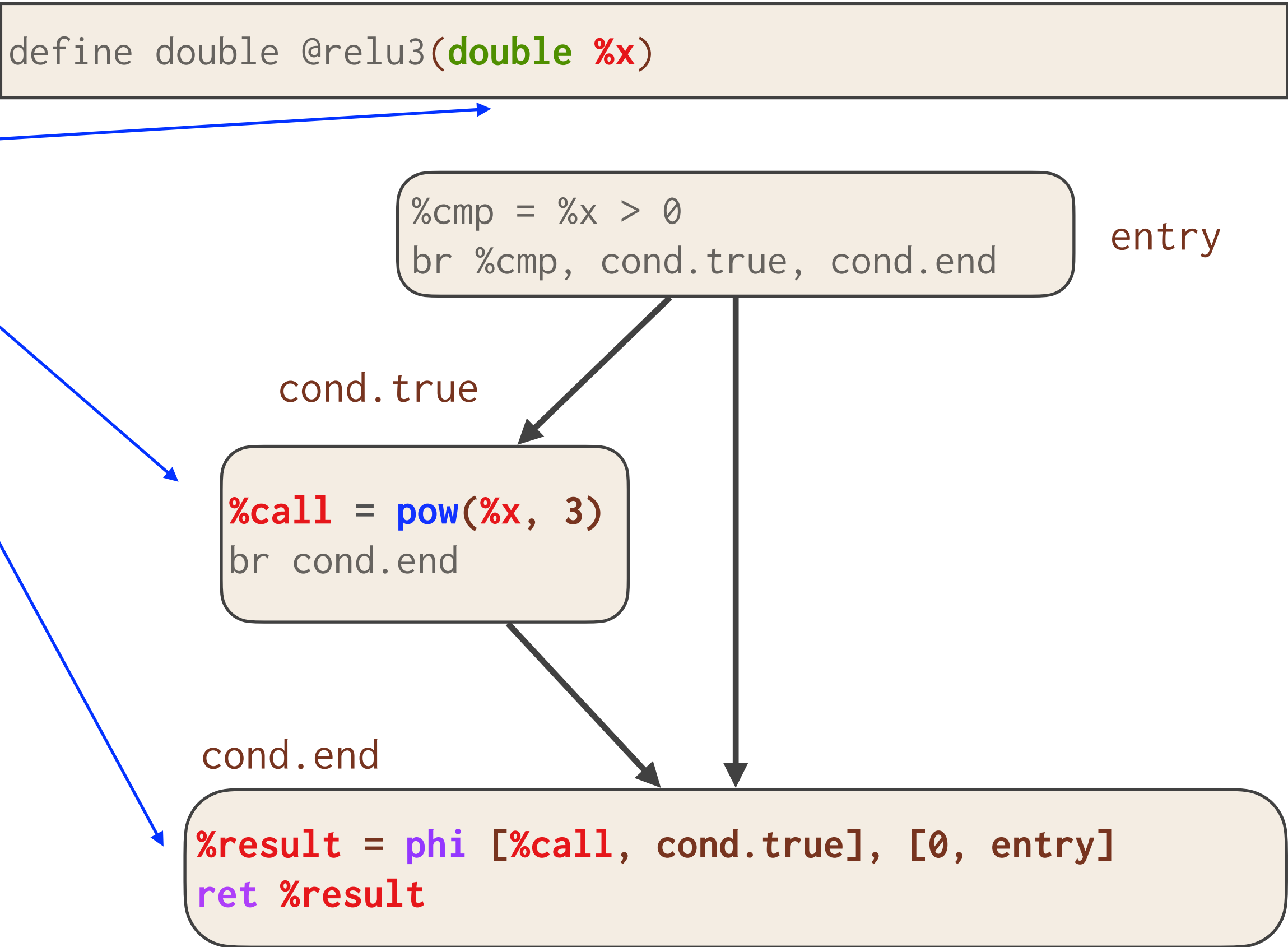
```
double diffe_relu3(double x) {  
    return __enzyme_autodiff(relu3, x);  
}
```

LLVM

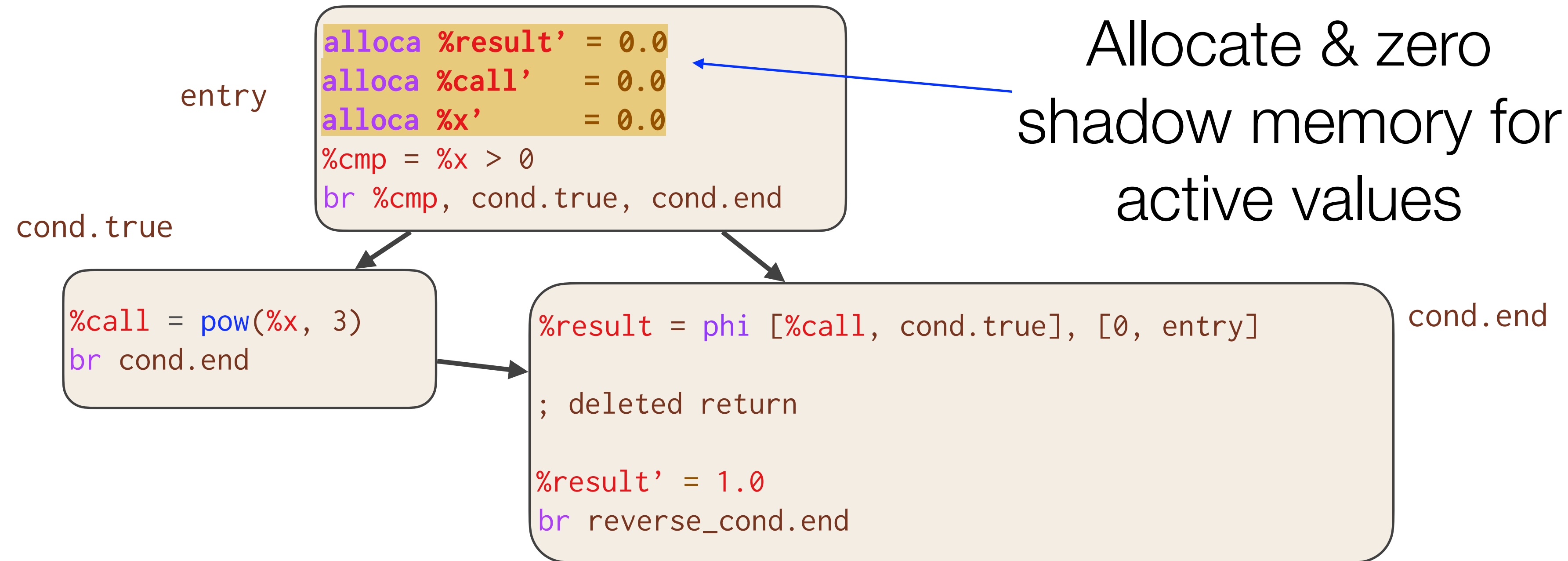


Case Study: ReLU3

Active Instructions

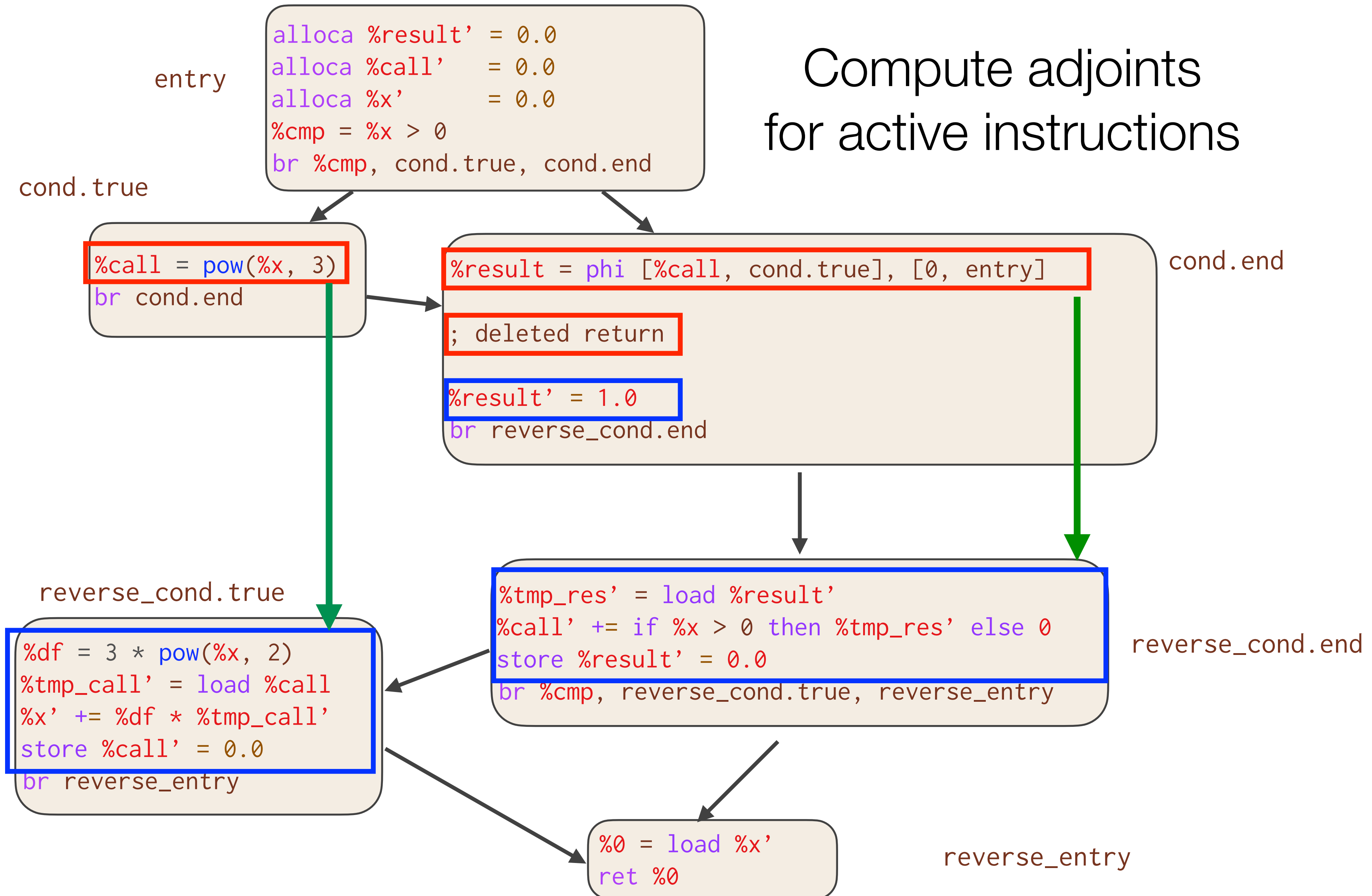


```
define double @diffe_relu3(double %x, double %differet)
```



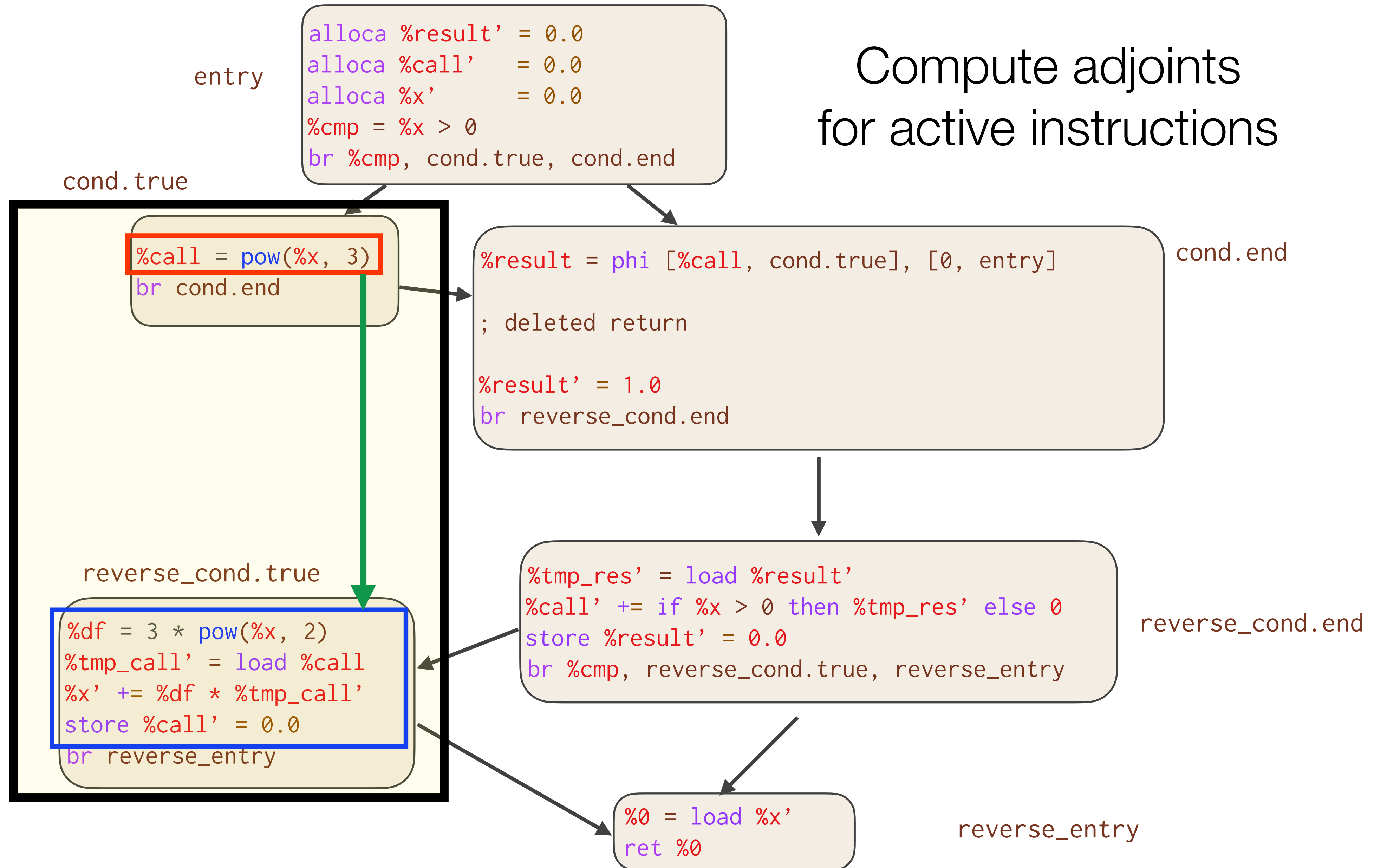
```
define double @diffe_relu3(double %x, double %differet)
```

Compute adjoints for active instructions



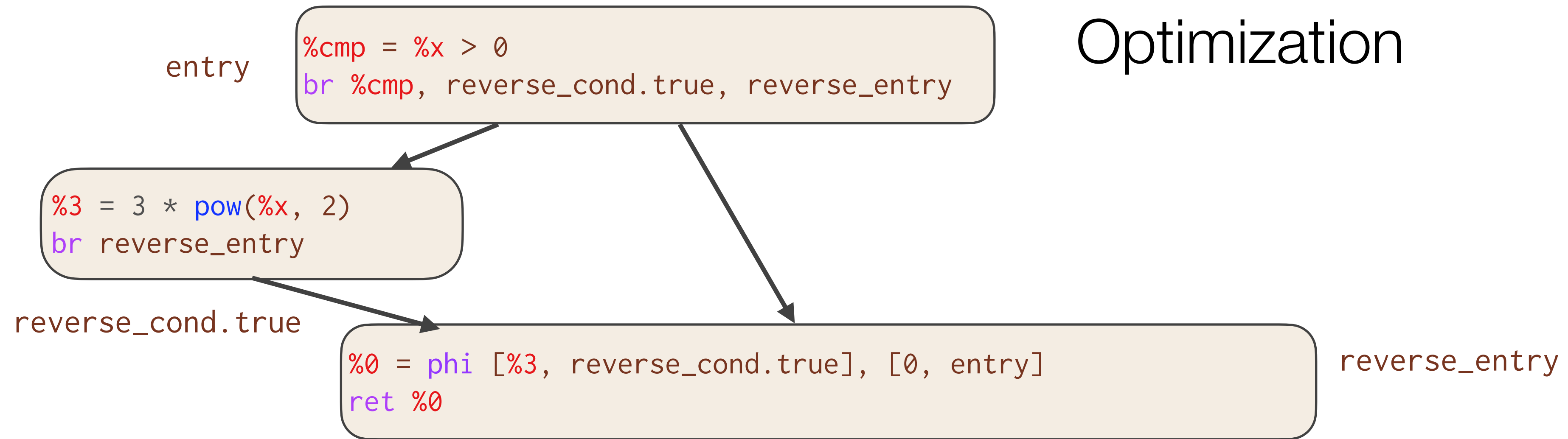
```
define double @diffe_relu3(double %x, double %differet)
```

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;  
}
```



Challenges of Low-Level AD

- Low-level code lacks information necessary to compute adjoints

```
void f(void* dst, void* src) {  
    memcpy(dst, src, 8);  
}
```

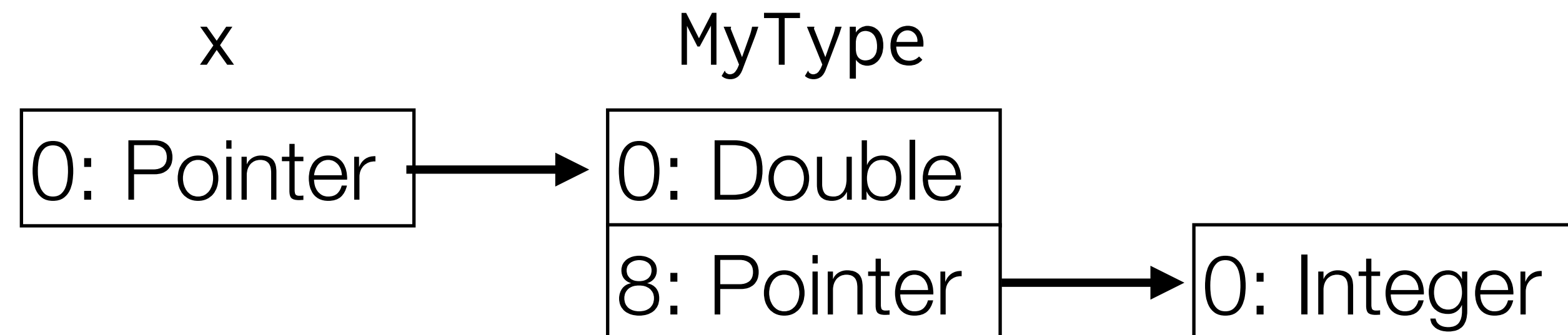
```
void grad_f(double* dst, double* dst',  
            double* src, double* src') {  
    // Forward Pass  
    memcpy(dst, src, 8);  
  
    // Reverse Pass  
    src'[0] += dst'[0];  
    dst'[0] = 0;  
}
```

```
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*;
```



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

Case 3: Store, Sync, Store

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

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

sync_threads;

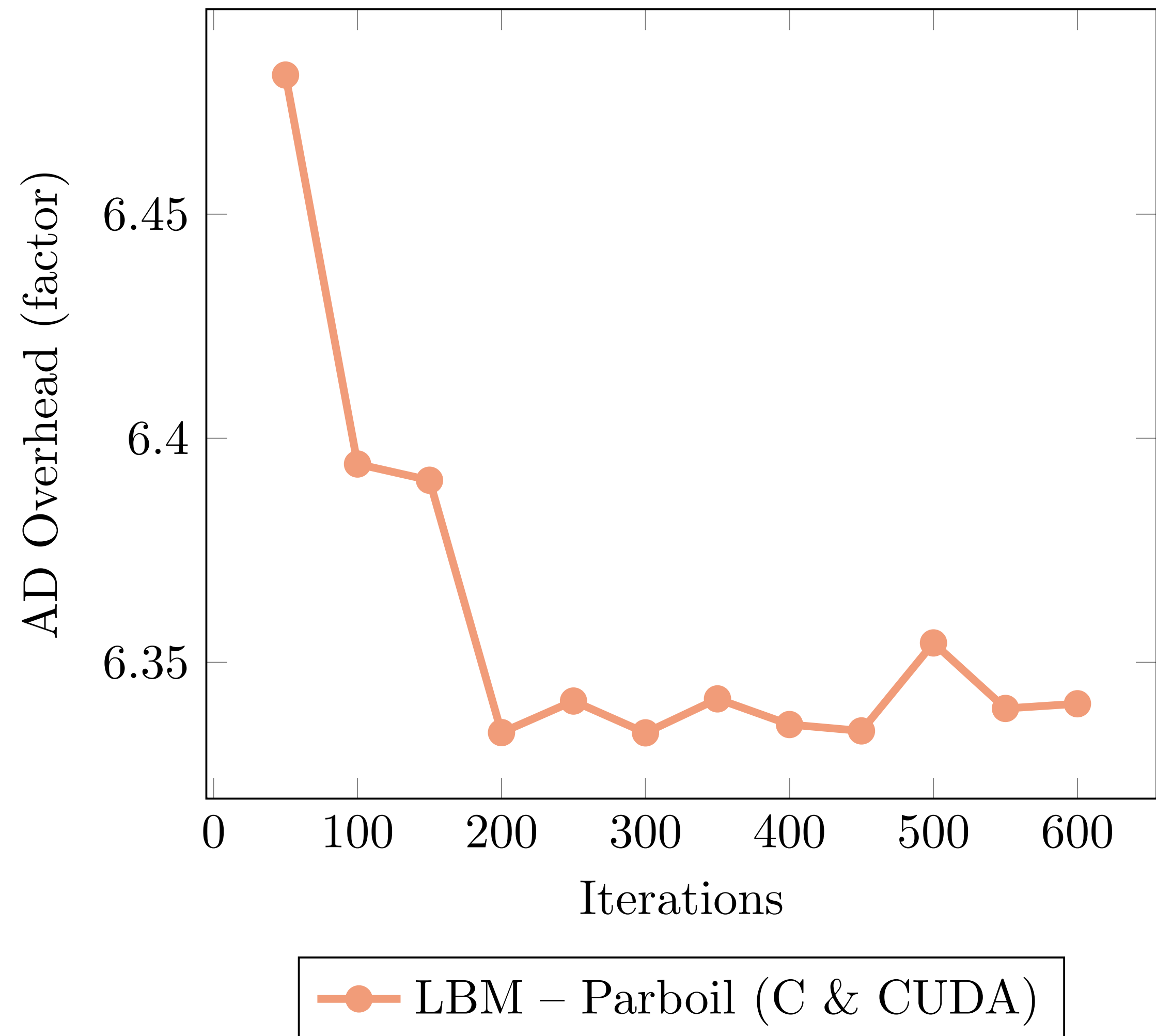
diffe_codeA(); // load %d_ptr
                // store %d_ptr = 0
```



Correct

- 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

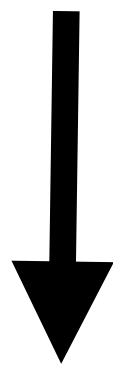
```
double grad_square(double val) {  
    return 2 * val;  
}
```

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



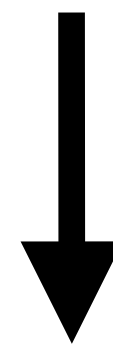
Parallel Automatic Differentiation in LLVM

```
%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)

