Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients!


William S. Moses   Valentin Churavy

wmoses@mit.edu

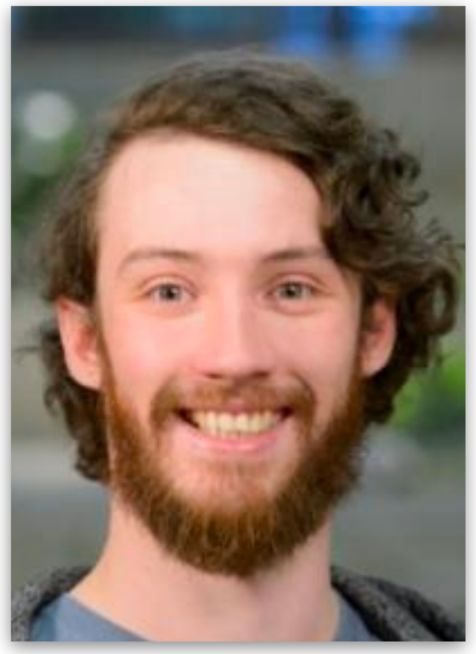
June 23, 2021

Argonne NATIONAL LABORATORY

William S. Moses

Valentin Churavy

Ludger Paehler



Johannes Doerfert

Jan Hückelheim

Sri Hari Krishna Narayanan

Michel Schanen

Paul Hovland

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

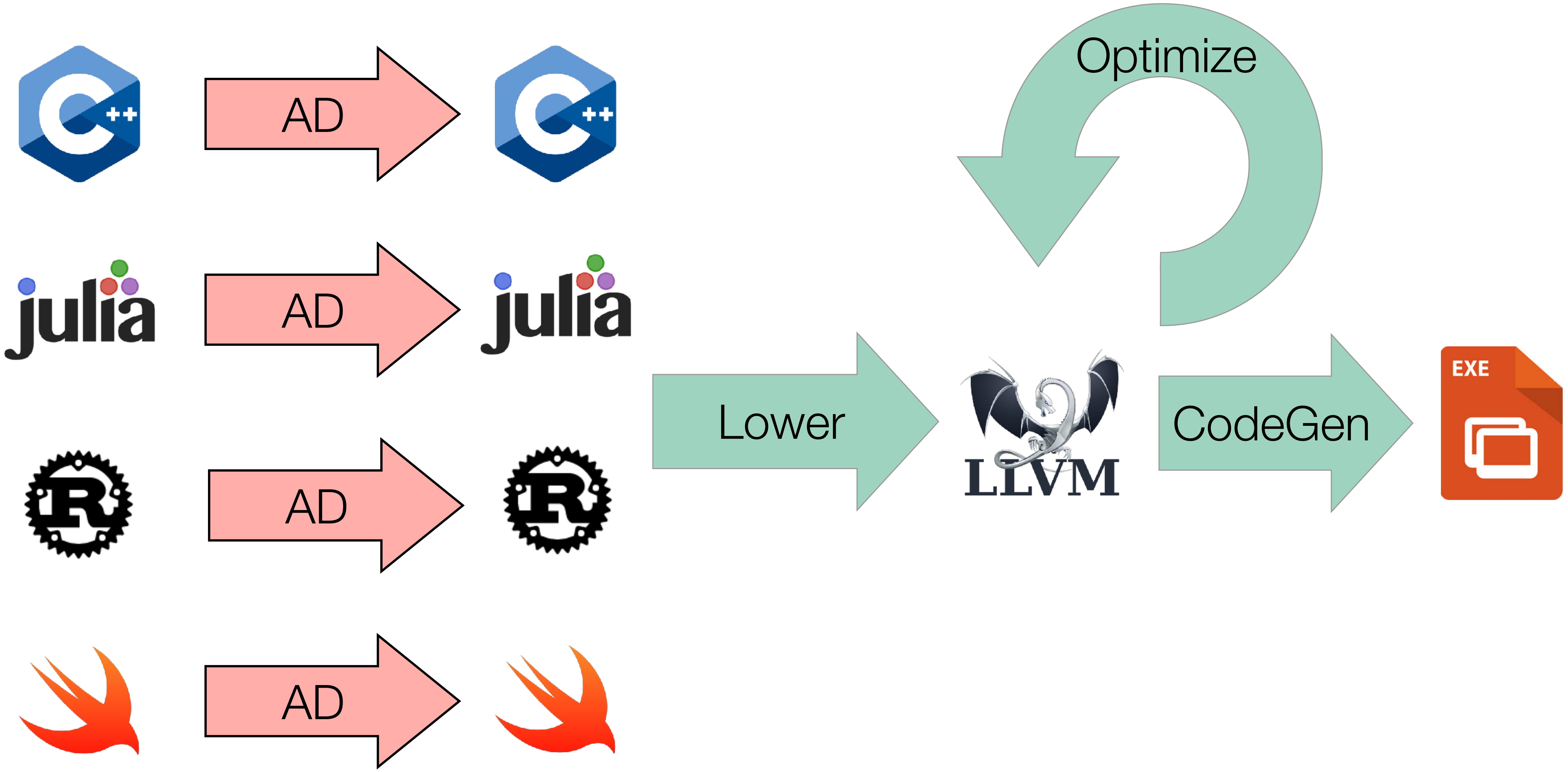
Existing AD Approaches

- 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

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

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

```
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)
}
```

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)
}
```

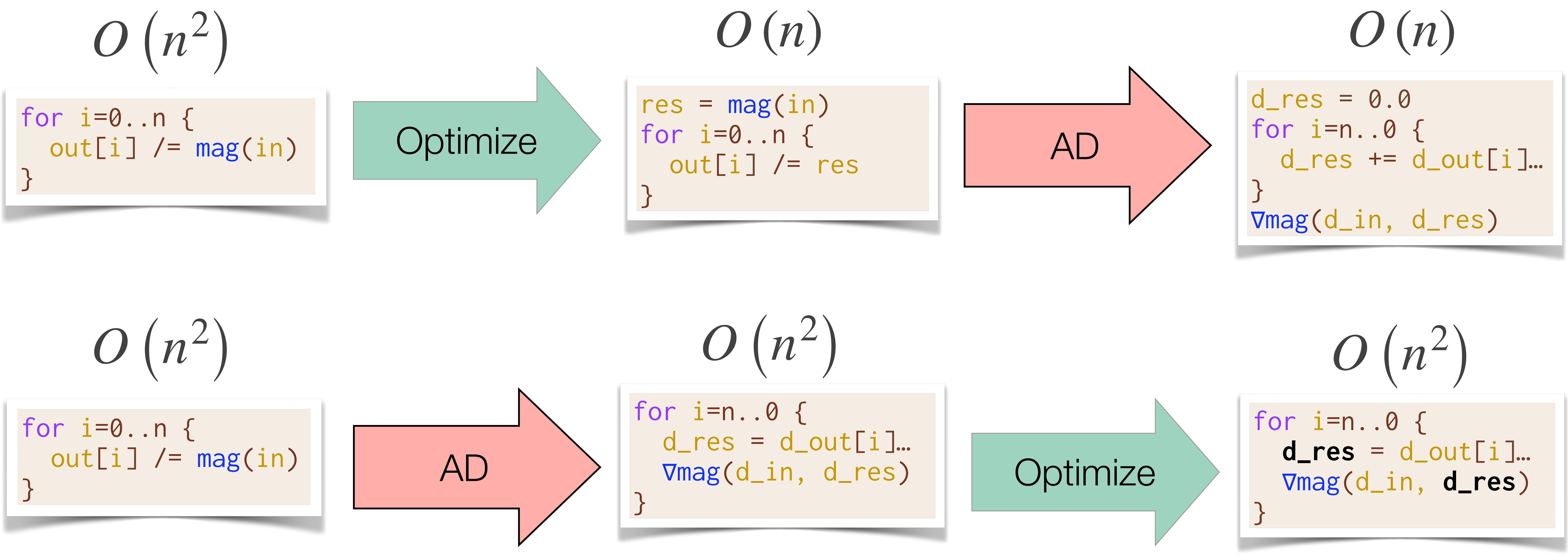
Optimize

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

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

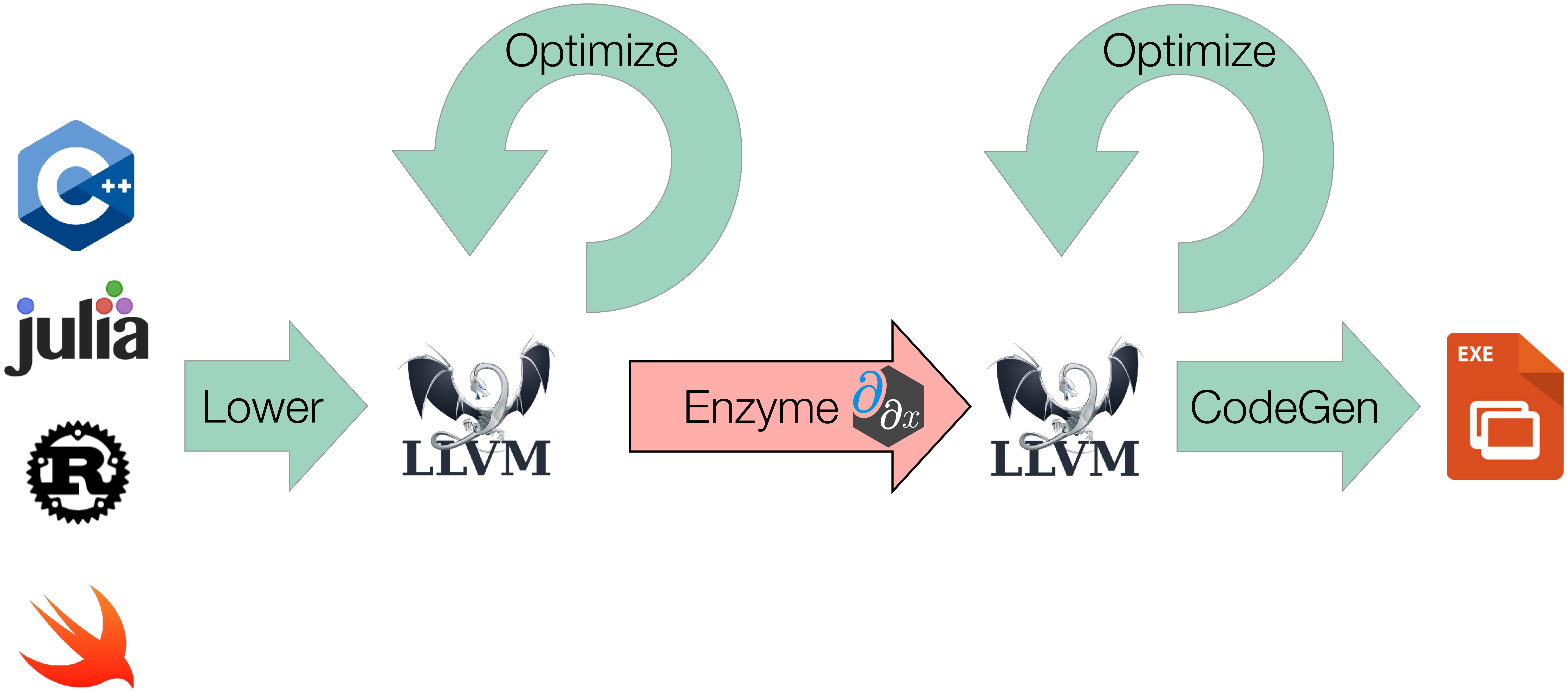
Optimization & Automatic Differentiation

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



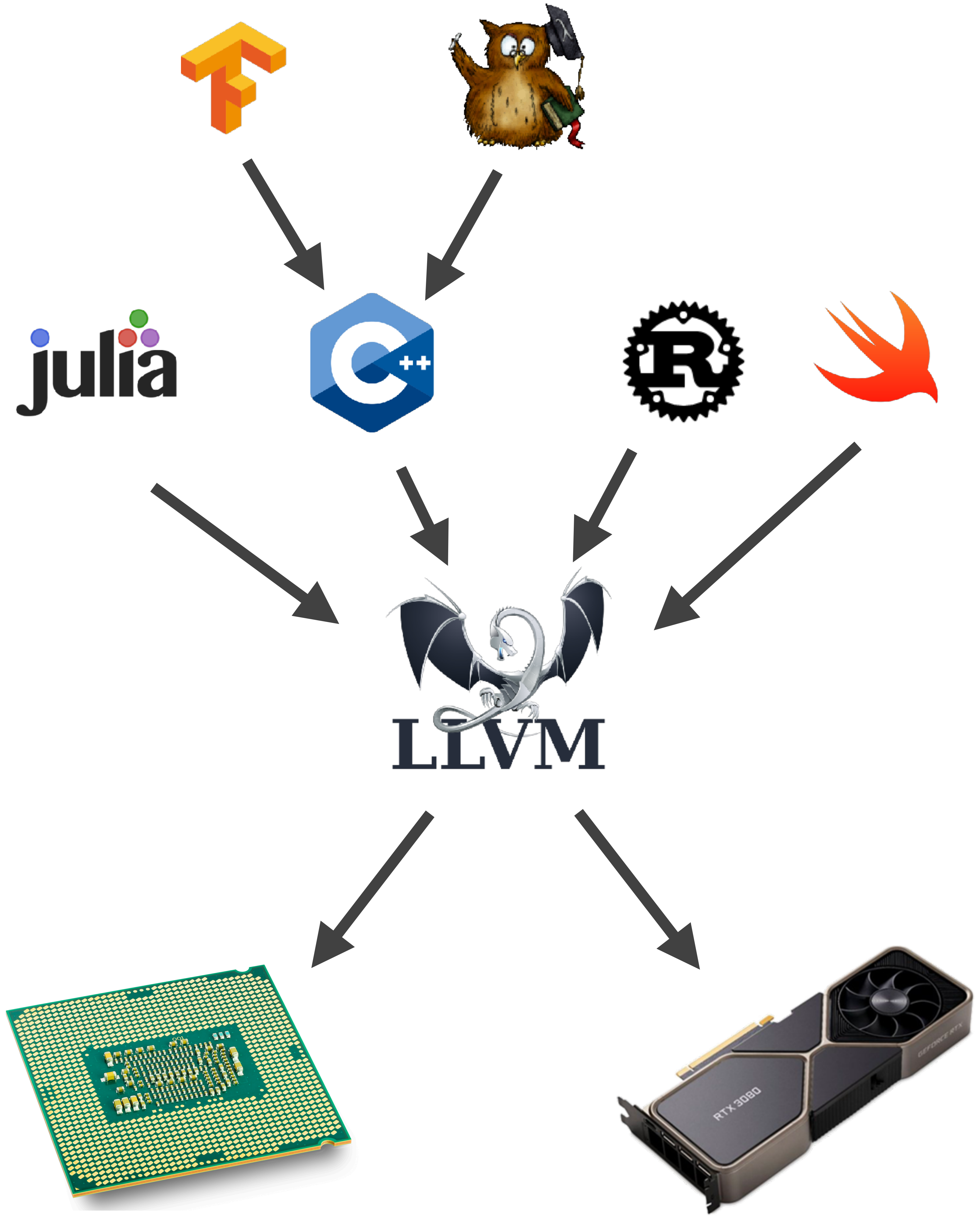
Enzyme Approach

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



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



Case Study: ReLU3

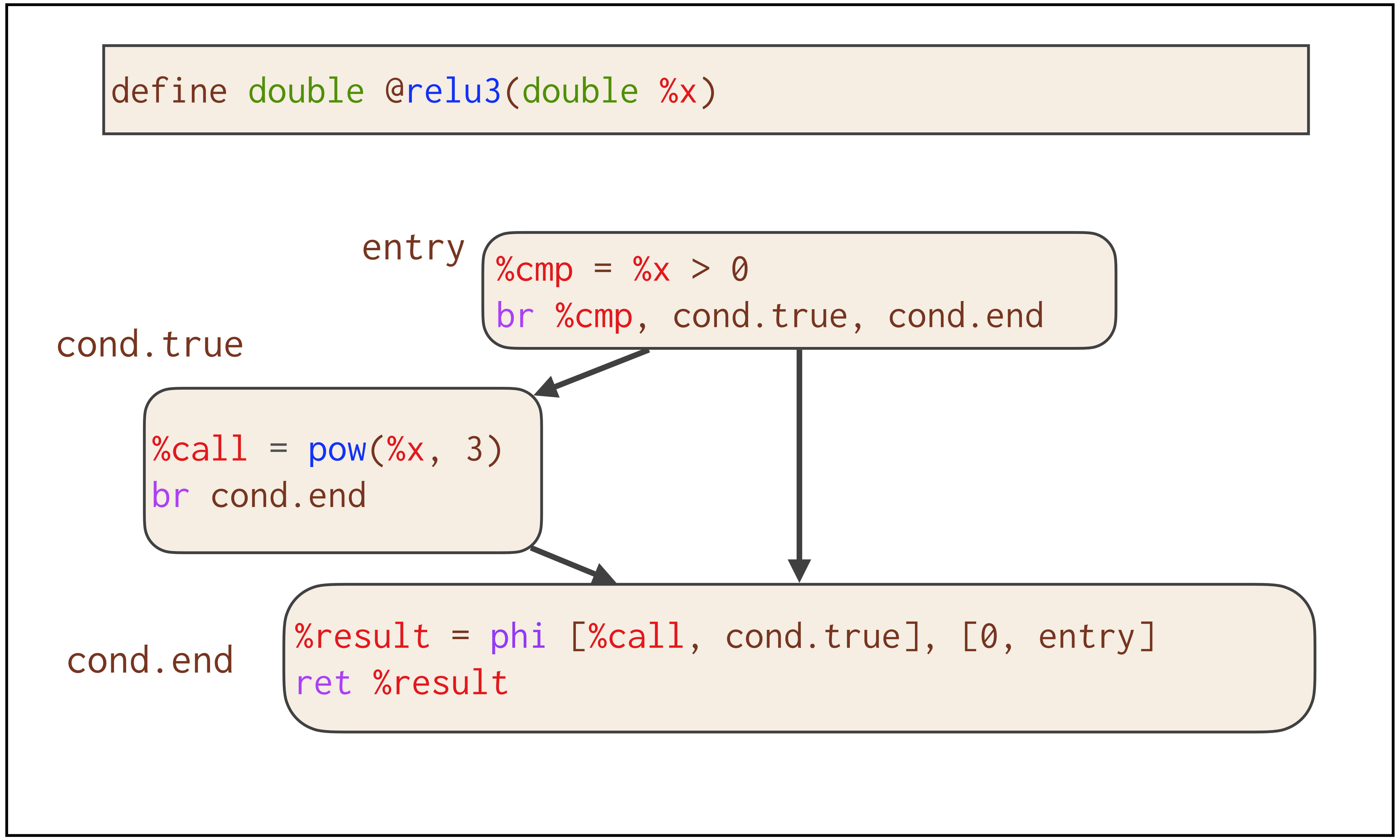
C Source

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

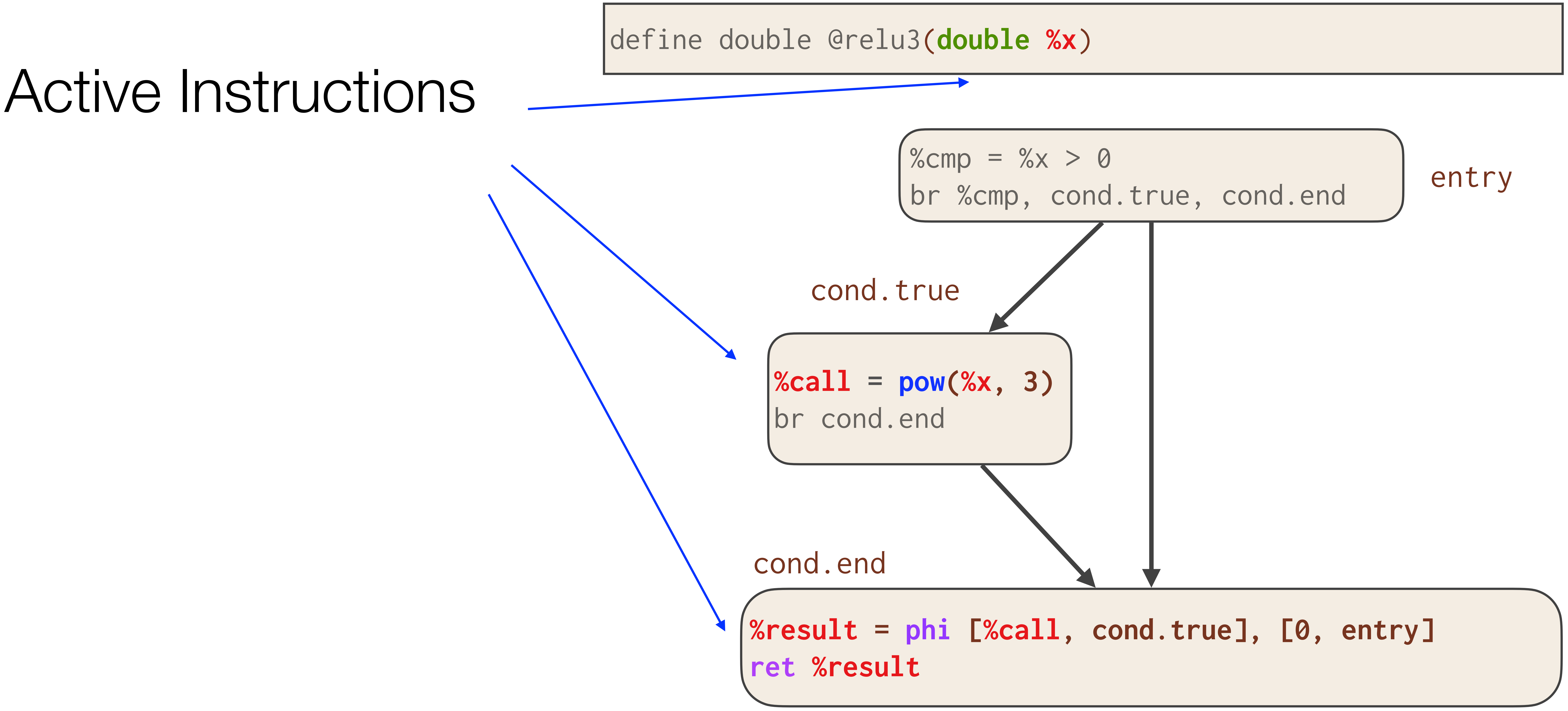
Enzyme Usage

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

LLVM



Case Study: ReLU3



```
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

```
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
```

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
```

reverse_cond.end

```
%tmp_res' = load %result'
%call' += if %x > 0 then %tmp_res' else 0
store %result' = 0.0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true

```
%df = 3 * pow(%x, 2)
%tmp_call' = load %call
%x' += %df * %tmp_call'
store %call' = 0.0
br reverse_entry
```

reverse_entry

```
%0 = load %x'
ret %0
```

Compute adjoints for 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
```

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
```

reverse_cond.end

```
%tmp_res' = load %result'
%call' += if %x > 0 then %tmp_res' else 0
store %result' = 0.0
br %cmp, reverse_cond.true, reverse_entry
```

reverse_cond.true

```
%df = 3 * pow(%x, 2)
%tmp_call' = load %call
%x' += %df * %tmp_call'
store %call' = 0.0
br reverse_entry
```

reverse_entry

```
%0 = load %x'
ret %0
```

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

Post Optimization

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!

```
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);
}
```

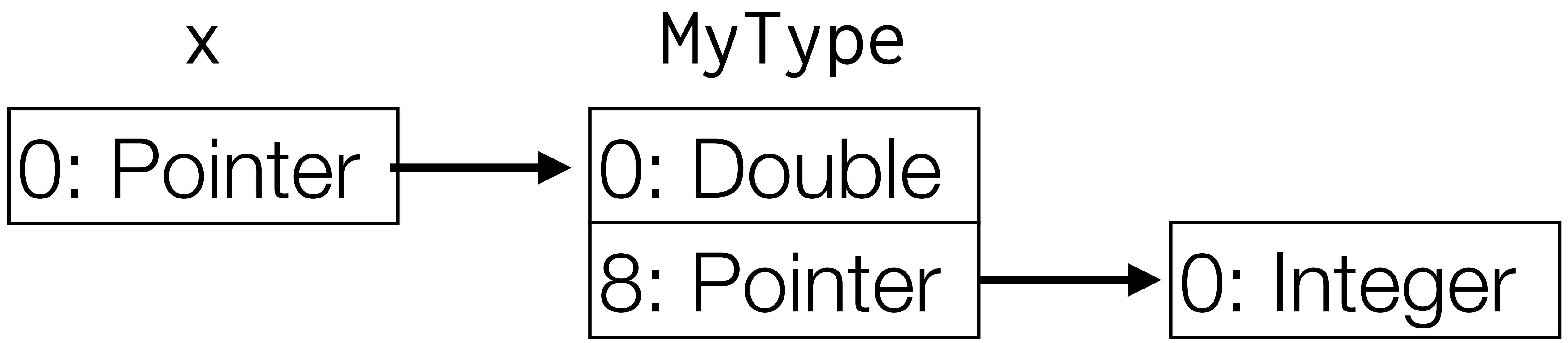
```
void grad_f(double* dst, double* dst',
            double* src, double* src') {
  // Forward Pass
  memcpy(dst, src, 8);

  // 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*;
```



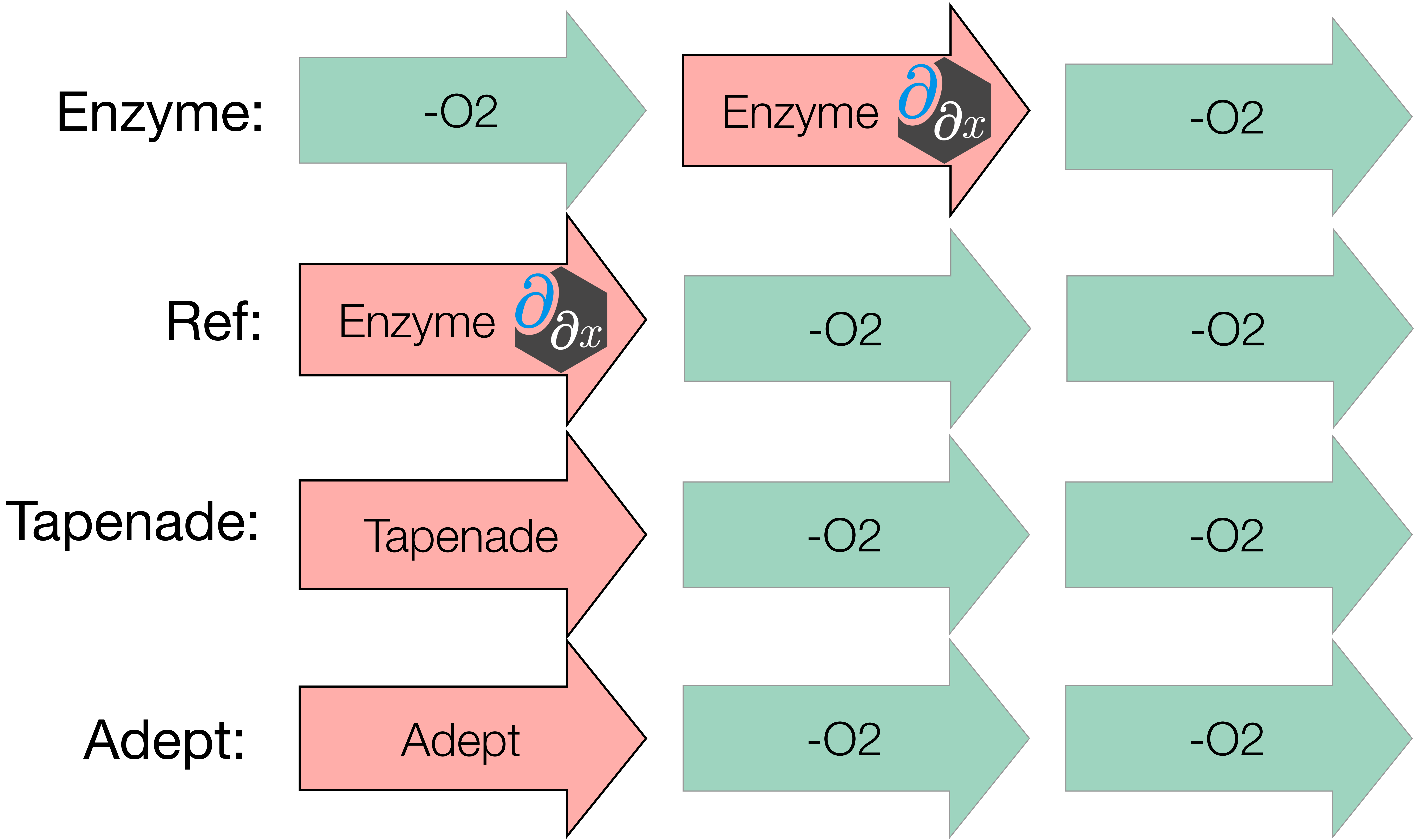

```
types(x) = {[0]:Pointer, [0,0]:Double, [0,8]:Pointer, [0,8,0]:Integer}
```

Cache

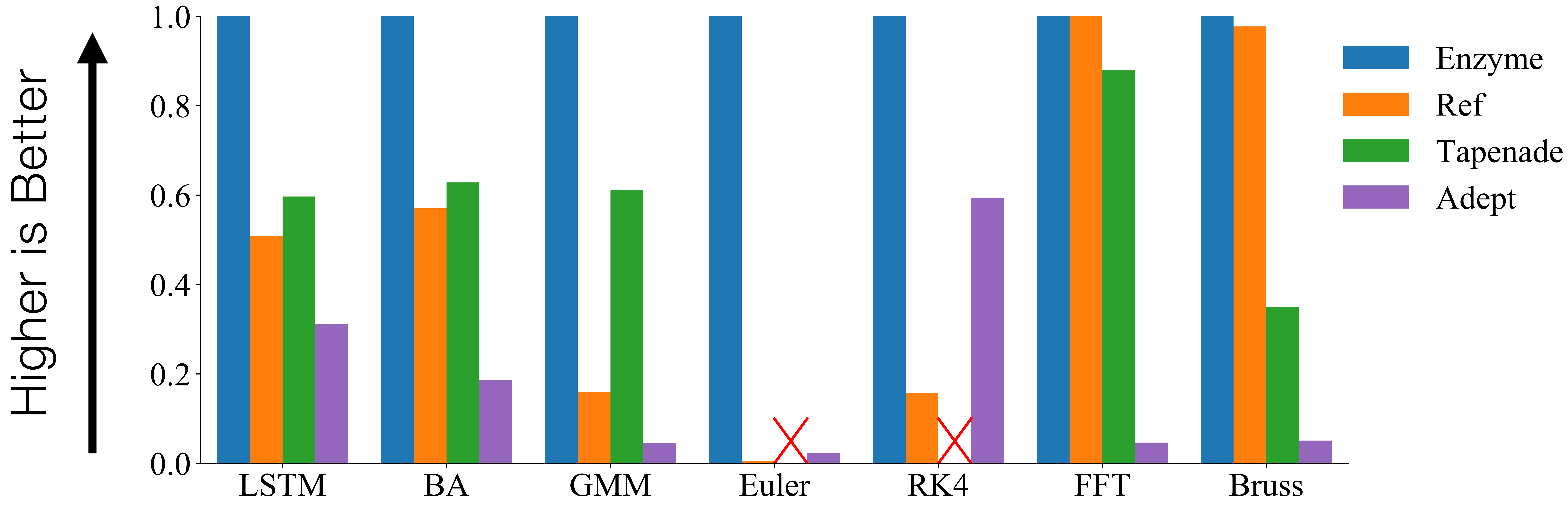
- Adjoint instructions may require values from the forward pass
    - e.g. $\nabla(x * y) \Rightarrow x\,dy + y\,dx$
- For all values needed in the reverse, allocate memory in the forward pass to store the value
- Values computed inside loops are stored in an array indexed by the loop induction variable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

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
  - Reversing parallel control flow can lead to incorrect results
  - Complex performance characteristics make it difficult to synthesize efficient code
  - 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)

```
void set(double* ar, double val) {

  parallel_for(int i=0; i<10; i++)
    ar[i] = val;
}
```

Read Race

Write 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;
}
```

GPU Memory Hierarchy

Per Thread	Per Block	Per GPU
Register	Shared Memory	Global Memory
~Bytes	~KBs	~GBs
Use Limits Parallelism	Use Limits Parallelism	

Slower, larger amount of memory

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
```

✓ Correct

- 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) {
  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; }
}
```

Efficient GPU Code

- Without optimization, GPU gradients must cache a large number of values
    - 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 LLVM optimizations can reduce the overhead
- Unlike the CPU, existing LLVM optimizations aren’t sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude

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]);
  }
}
```

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

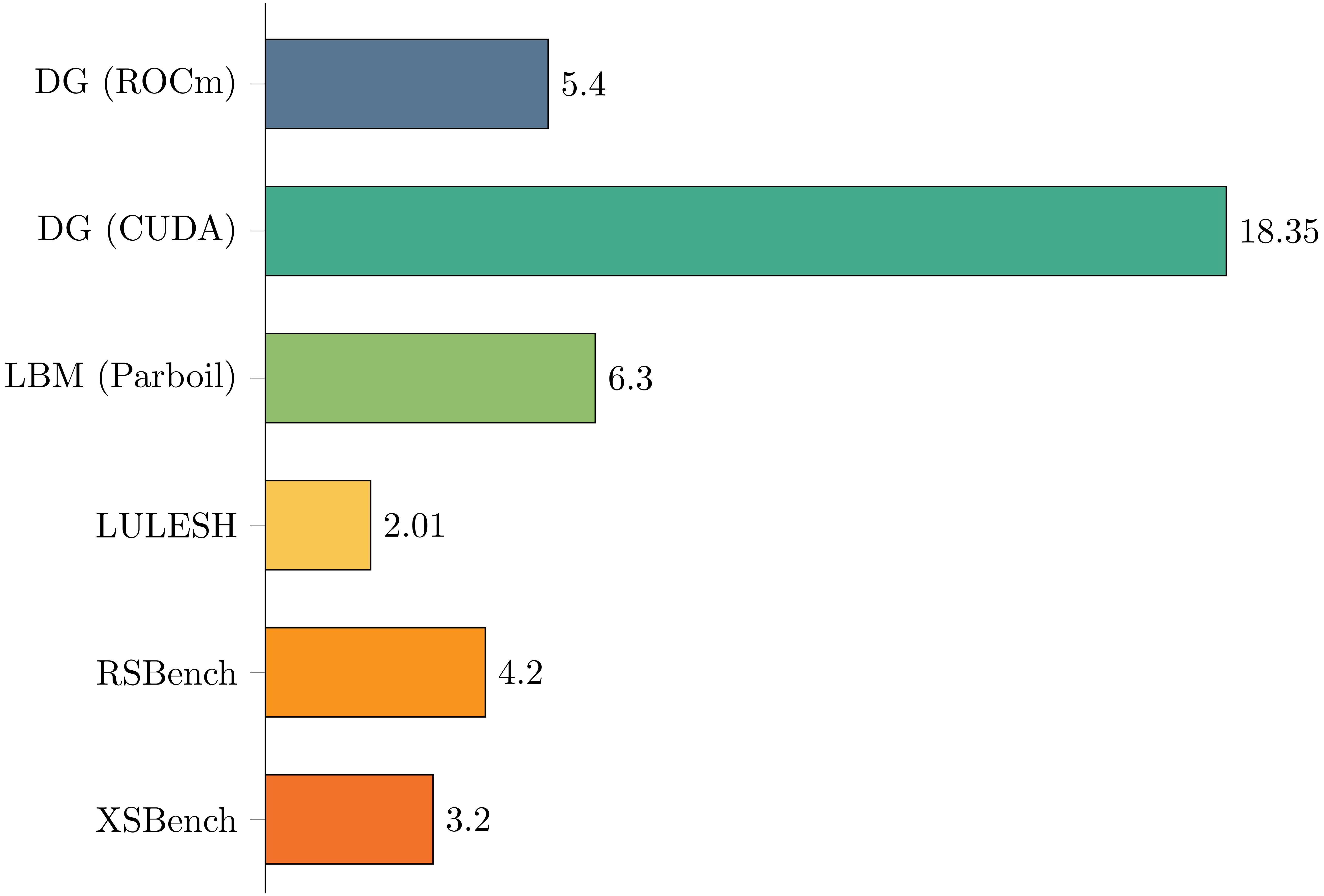
- Not all values are needed for the reverse pass. By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.

  - 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);
```

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

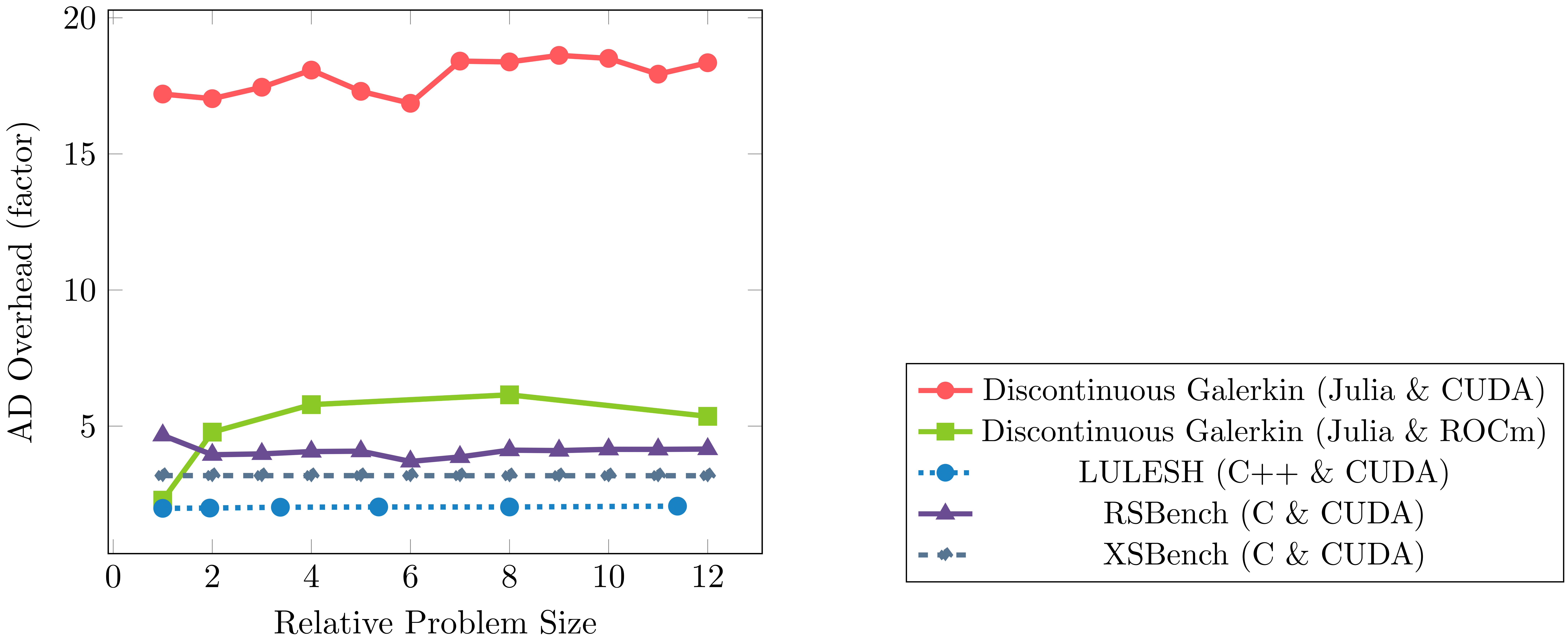
Ablation Analysis of Optimizations

Ablation Analysis of Optimizations

Ablation Analysis of Optimizations

GPU AD is Intractable Without Optimization!

Scalability Analysis (Fixed Work Per Thread)



Enzyme + Legion

- Caching within automatic differentiation requires solving a data availability problem for maximum performance
- Enzyme contains utilities to analyze both the serial and parallel dependency structure of the program and contains several optimizations to locally reduce cache sizes
- Presently, Enzyme keeps the schedule for the original program and for both the augmented forward and reverse pass
- Can we leverage Legion to analyze the dependence structure, develop a minimum cache using domain-specific information, and provide high performance (and perhaps distinct) mappings for the forward and reverse pass?

Enzyme

- 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 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
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow

Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Hal Finkel, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Miguel Young de la Sota, 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.

- This research was supported in part by LANL grant 531711. 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.

Enzyme + Legion

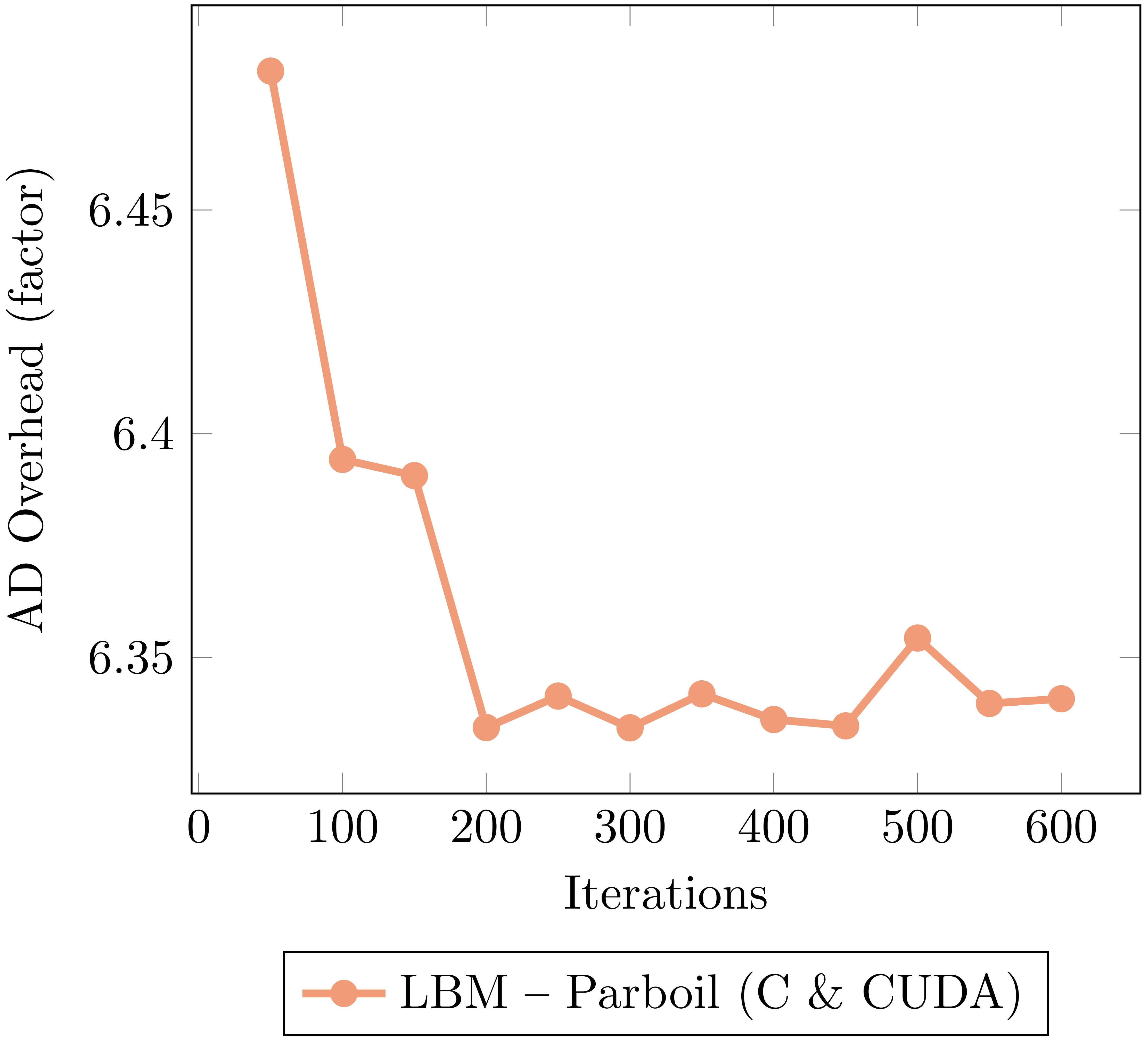
- Caching within automatic differentiation requires solving a data availability problem for maximum performance
- Enzyme contains utilities to analyze both the serial and parallel dependency structure of the program and contains several optimizations to locally reduce cache sizes
- Presently, Enzyme keeps the schedule for the original program and for both the augmented forward and reverse pass
- Can we leverage Legion to analyze the dependence structure, develop a minimum cache using domain-specific information, and provide high performance (and perhaps distinct) mappings for the forward and reverse pass?

Enzyme

- 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 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
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow

Enzyme

Scalability Analysis (Fixed Thread Count)



PyTorch-Enzyme & TensorFlow-Enzyme

```python
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)
```

```python
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)
```

```c
// 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);
}
```

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

```
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

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

```
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
```

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

Allocate & zero shadow memory per active value

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

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
```

Cache forward pass variables for use in reverse

```
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
```

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
```

for.cleanup

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

After lowering & some optimizations

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

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

Enzyme

- 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

CUDA Automatic Differentiation

```
%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.

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

Enzyme

- 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

Differentiation Is Key To Machine Learning

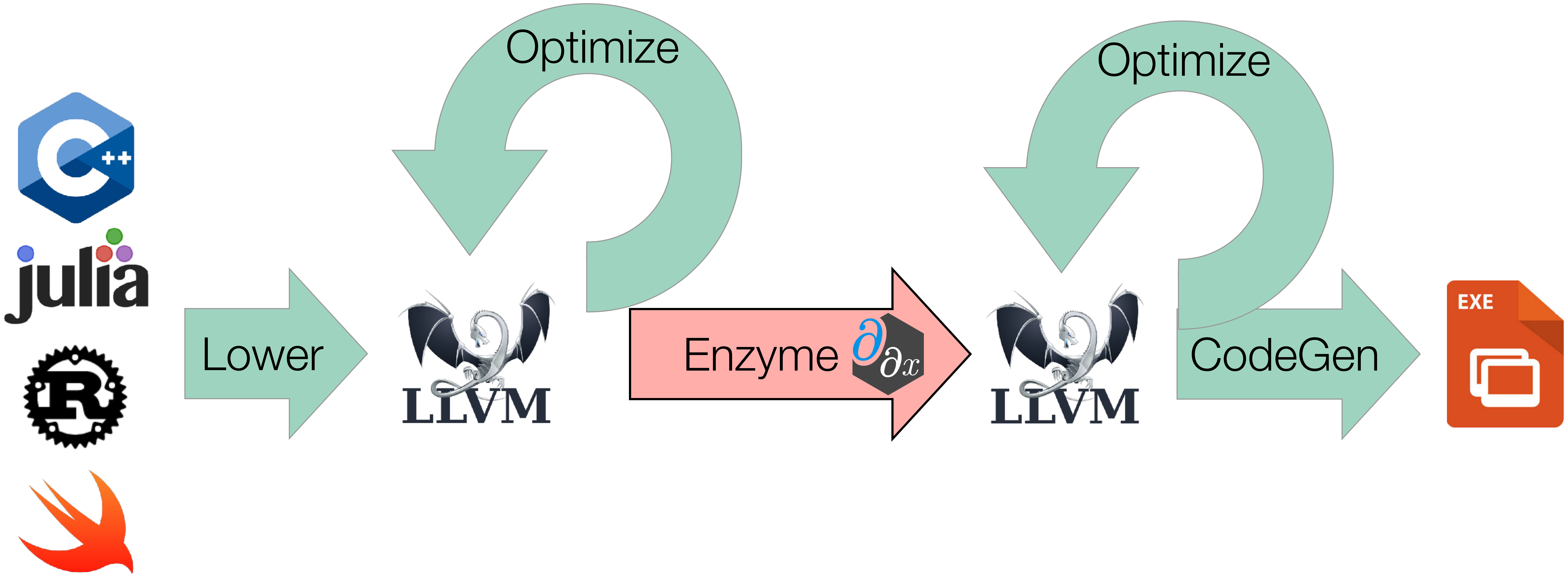
```cpp
// C++ nbody simulator

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;
  }
}
```

```python
// 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
```

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

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]
  - PHI Node unwrapping

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

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