Enzyme: High-Performance Automatic Differentiation of LLVM

William S. Moses  Valentin Churavy

wmoses@mit.edu
LLVM Developers’ Meeting
October 8, 2020
Automatic Differentiation

- Many algorithms require the derivatives of various functions
  - 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 automatically create derivatives automatically
Existing AD Approaches

• Differentiable DSL (TensorFlow, PyTorch, DiffTaichi)
  • Provide a new language where all functions are differentiable
  • 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
  - Requires all code to be available ahead of time
  - Difficult to use with external libraries
Existing AD Pipelines

- **CodeGen**
- **Optimize**
- **Lower**
- **LLVM**
- **EXE**
Case Study: Vector Normalization

//Compute magnitude in O(n)
double mag(double* x, size_t n);

//Compute norm in O(n^2)
void norm(double* out, double* in, size_t n) {
    for(int i=0; i<n; i++) {
        out[i] = in[i]/mag(in, n);
    }
}
double mag(double* x, size_t n);

void norm(double* out, double* in, size_t n) {
    double res = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }
}

Loop Invariant Code Motion

O(n^2)
LICM then Differentiate

```c
void dnorm(double* out, double* dout,
            double* in, double* din, size_t n) {
    double res = mag(in, n);
    for (int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }
    double d_res = 0;
    for (int i=0; i<n; i++) {
        dres += -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
    }
    dmag(in, din, n, dres);
}
```

$O(n)$

$O(n)$
void dnorm(double* out, double* dout, double* in, double* din, size_t n) {
    for(int i=0; i<n; i++) {
        out[i] = in[i]/mag(in, n);
    }
    for(int i=0; i<n; i++) {
        double dres = -in[i]*in[i]/mag * dout[i];
        din[i] += dout[i]/mag;
        dmag(in, din, n, dres);
    }
}

\[ O(n^2) \]

Just Differentiate
Differentiate then LICM

```c
void dnorm(double* out, double* dout,
            double* in, double* din, size_t n) {

    double res = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }

    for(int i=0; i<n; i++) {
        double dres = -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
        dmag(in, din, n, dres);
    }
}
```

Can’t LICM as dmag uses loop-local variable dres
Enzyme Approach

Perform AD on **optimized** programs!
How to Achieve Post-Optimization AD

• Implement all optimizations in AD system
  • Embed a compiler into your AD
  • Rewrite all compiler analyzes and optimizations
• Perform AD on low-level post-optimization representation
  • Embed AD into your compiler
• “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
Enzyme

- Reverse-mode source-rewriting AD plugin for statically analyzable LLVM IR
- 4.5x speedup over AD before optimization
- State-of-the art performance with existing tools
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- PyTorch-Enzyme & TensorFlow-Enzyme packages let researchers use foreign code in their ML workflow
- Multisource AD & library support by leveraging LTO
Why LLVM?

- Generic low-level compiler infrastructure with many frontends
  - “Cross platform assembly”
- Well-defined semantics
- Large collection of optimizations
- Analysis passes can be used as utilities
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
The “memcpy” Problem

• Taking the derivative of operations such as memcpy depends on the type of the data being copied
  • e.g. one derivative for pointers, one for doubles, another for floats
• LLVM Types != C/C++ types
Case Study: Read Sum

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

```c
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;
}
```

```c
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

```c
struct Type {
    double;
    int*;
}
x = Type*;
```

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

- Initialize type trees
  - Constant, TBAA, and known instructions
- Perform series of fixed-point updates
  - Each instruction has a type propagation rule
    \[
    z\{\} = \text{add } x\{0:\text{Int}\}, y\{0:\text{Int}\}
    \]
    \[
    \downarrow
    \]
    \[
    z\{0:\text{Int}\} = \text{add } x\{0:\text{Int}\}, y\{0:\text{Int}\}
    \]
- Provide a compile-time error if a necessary type cannot be deduced statically
Type Analysis

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
void callee(int* ptr) {
    ptr: {[0]:Pointer, [0,16]:Double, [0,24]:Int}
    ptr2: {[0]:Pointer, [0,0]:Double, [0,8]:Int}
    loadtype: {[0]:Double}
    ptr3: {[0]:Pointer, [0,0]:Int}
    cptr2: {[0]:Pointer, [0,0]:Double, [0,8]:Int}
    notype: {[0]:Double}
    cptr3: {[0]:Pointer, [0,0]:Int}
}
```
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
Shadow Memory

- Derivatives of values are stored in shadow allocations

- For all active values, allocate and zero shadow memory to store the derivative of all of its occurrences

- All data structures need to have a shadow data structure created
  - Enzyme will create shadow allocation/stores for structures created inside code being differentiated
  - Data structures passed as arguments will pass shadow arguments
Derivative Synthesis

- Initialize shadow memory

- For each BasicBlock BB:
  
  - For each Instruction I in reverse(BB):
    
      - Emit adjoint I, caching and reloading any necessary values from the forward pass
Case Study: ReLU3

define double @relu3(double %x)

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

double diffe_relu3(double x) {
  return __enzyme_autodiff(relu3, x);
}
Case Study: ReLU-f

Active Instructions

define double @relu3(double %x)

%cmp = %x > 0
br %cmp, cond.true, cond.end

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

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

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

%result = phi [%call, cond.true], [0, entry]
; deleted return
%result' = 1.0
br reverse_cond.end

cond.true

cond.end

Allocate & zero shadow memory for active instructions
```
Compute adjoints for active instructions

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

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

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

%result = phi [%call, cond.true], [0, entry]
%result’ = 1.0
br reverse_cond.end

alloca %result
alloca %call’
alloca %x’
%cmp = %x > 0
br %cmp, reverse_cond.true, reverse_entry

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

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

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

%0 = load %x’
ret %0
```

define double @diffe_relu3(double %x)

Essentially the optimal hand-compiled program!

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

- Adjoint instructions may require values from the forward pass
  - e.g. $\nabla(x \times y) => x \ dy + y \ dx$
- For all such values, allocate memory in the function header to store the value for use in the reverse pass
- 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

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

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

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

```
%result = phi [ %call, cond.true], [0, entry] 
ret %result
```
Case Study: Read Sum

define double @sum(double* %x)

%result = phi [%call, cond.true], [0, entry]  
ret %result
Each register in the for loop represents a distinct active variable every iteration.
Define double @diffe_sum(double* %x, double* %xp)

Allocate & zero shadow memory per active value

Entry

Allocate & zero shadow memory per active value

For.body

For.cleanup

Return %result
define double @diffe_sum(double* %x, double* %xp)

Cache forward pass variables for use in reverse

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
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

After lowering & some optimizations
Case Study: Read Sum

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

%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

```c
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();
}
```
Cache Optimizations

• By carefully caching in a form LLVM understands, existing optimization passes can optimize the memory away! [*]

• Further optimizations:
  • Use alias analysis to prove that recomputing an instruction is legal
  • Don’t cache unnecessary values
  • Don’t cache a value that already has already been cached elsewhere

[*] For dynamic loop optimizations, LLVM must understand semantics of realloc.
Function Calls

- Computing both forward and reverse pass in the same function allows further optimization and reduces memory usage
  - Enzyme uses Alias Analysis to detect legality of computing forward/reverse pass together
  - Otherwise, Enzyme may need to modify forward pass to cache values needed by reverse pass
Indirect Function Calls

- Calls to functions that aren’t known at compile time are dealt with by leveraging shadow memory

- The shadow of function pointers is defined to be a global containing the forward and reverse pass

- Thus taking the adjoint of an indirect function call simply requires extracting and calling the corresponding shadow callee
Custom Derivatives & Multisource

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

```c
__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
Evaluation

- Collection of benchmarks from Microsoft’s ADBench suite and of technically interest
- Evaluated Enzyme, Reference, and the two fastest AD systems from ADBench (Tapenade, Adept)
- All programs run serially
- Quiesed Amazon c4.8xlarge (disabled turbo-boost; hyper-threading)
Experimental Setup

Enzyme: -O2

Ref: Enzyme $\delta\partial_x$ -O2

Tapenade: Tapenade -O2

Adept: Adept -O2
Relative Speedup

Higher is Better

Speedup of 0.5 denotes program took twice as long as Speedup of 1.0
## Runtime

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Ref</th>
<th>Tapenade</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>2.353</td>
<td>4.458</td>
<td>4.042</td>
<td>7.645</td>
</tr>
<tr>
<td>BA</td>
<td>0.424</td>
<td>0.778</td>
<td>0.680</td>
<td>2.334</td>
</tr>
<tr>
<td>GMM</td>
<td>0.073</td>
<td>0.462</td>
<td>0.124</td>
<td>1.544</td>
</tr>
<tr>
<td>Euler</td>
<td>0.161</td>
<td>36.723</td>
<td>nan</td>
<td>6.851</td>
</tr>
<tr>
<td>RK4</td>
<td>3.397</td>
<td>23.442</td>
<td>nan</td>
<td>6.371</td>
</tr>
<tr>
<td>FFT</td>
<td>0.183</td>
<td>0.182</td>
<td>nan</td>
<td>2.538</td>
</tr>
<tr>
<td>Bruss</td>
<td>0.181</td>
<td>0.182</td>
<td>0.518</td>
<td>3.457</td>
</tr>
</tbody>
</table>

Enzyme is 4.5x faster than Ref!
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

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 diffe(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);
}
Conclusions

- AD on low-level IR can be fast

- Optimization before AD is crucial

- Enzyme provides high-performance cross-language AD

- Open-sourcing now (visit enzyme.mit.edu & join our mailing list)
  - Hope to upstream as LLVM project

- Future Work:
  - Parallelism, GPU AD
  - AD-specific optimizations
Acknowledgements

• Thanks to James Bradbury, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Yo Shavit, Dhash Shrivathsa, Nalini Singh, and Alex Zinenko

• William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323.

• 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.
Conclusions

- AD on low-level IR can be fast
- Optimization before AD is crucial
- Enzyme provides high-performance cross-language AD
- Open-sourcing now (visit enzyme.mit.edu & join our mailing list)
  - Hope to upstream as LLVM project
- Future Work:
  - Parallelism, GPU AD
  - AD-specific optimizations
Backup Slides
Requirements & Performance Boosts

• Requirements
  • Enable TBAA (Type based alias analysis)
  • Strict Aliasing (no unions)
  • Disable exceptions

• Performance Boosts
  • Disable Loop Unrolling before AD
  • Disable Vectorization before AD
Future Work: Parallelism*

- Build off prior work [1] representing parallelism (OpenMP, Cilk, etc) in compiler
- Reverse pass can remain in parallel, with dependencies reversed
- Updates to adjoints in parallel tasks done with reducer or atomic add to prevent races

```c
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;
}
```

[*] Work in progress — suggestions appreciated
Benchmarks

• LSTM: Long-short term memory model
• BA: Bundle analysis
• GMM: Gaussian mixture model
• Euler: Euler integration
• RK4: Runge-Kutta integration
• FFT: Fast Fourier transform
• Bruss: Brusselrator chemical simulation
Matrix Vector: Single Iteration

- LLVM Optimization Passe
- Constant (wrt inputs) detection

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1.119</td>
<td>0.0006</td>
</tr>
<tr>
<td>Forward</td>
<td>1.119</td>
<td>11.016</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>1.210</td>
<td>13.445</td>
</tr>
</tbody>
</table>
Taylor Expan Log

LLVM Optimization Passe

Constant (wrt inputs) detection

```cpp
static adouble logger(adouble x) {
    adouble sum = 0;
    for (int i = 1; i <= ITERS; i++) {
        sum += pow(x, i) / i;
    }
    return sum;
}
```

```cpp
static double logger_and_gradient(double xin, double& xgrad) {
    adept::Stack stack;
    adouble x = xin;
    stack.new_recording();
    adouble y = logger(x);
    y.set_gradient(1.0);
    stack.compute_adjoint();
    xgrad = x.get_gradient();
    return y.value();
}
```
Taylor Expand Log (Julia)

- LLVM Optimization Passe

\[ f(x) = \sum_{i=1}^{N} \frac{x^i}{i} \approx -\log(1 - x) \]

- Constant (wrt inputs) detection

```c
#define ITERS 10000000

double logger(double x) {
    double sum = 0;
    for(int i=1; i<=ITERS; i++)
        sum += pow(x, i) / i;
    return sum;
}
```

```julia
function jl_f1(f::Float64)
    sum = 0 * f;
    for i = 1:10000000
        sum += f^i / i;
    end
    return sum;
end

f(x) ≈ \sum_{i=1}^{\infty} x^i / i 
≈ -\log(1 - x)
```

\[ \frac{\partial}{\partial x} f(x) \approx \frac{1}{1 - x} \]

\[ \frac{\partial}{\partial x} f(x = 0.5) \approx 2 \]

; Enzyme derivative code
@show autodiff(fl_f1, 0.5)
@time autodiff(fl_f1, 0.5)

using Zygote
@show jl_f1′(0.5)
@time jl_f1′(0.5)
Taylor Expand Log

- LLVM Optimization Passe
- Constant (wrt inputs) detection

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
<th>Enzyme-Julia</th>
<th>Zygote-Julia</th>
<th>AutoGrad-Julia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.74</td>
<td>3.72</td>
<td>3.82</td>
<td>3.82</td>
<td>3.82</td>
</tr>
<tr>
<td>Forward</td>
<td>3.74</td>
<td>4.56</td>
<td>3.82</td>
<td>3.82</td>
<td>3.82</td>
</tr>
<tr>
<td>Forward + Revers</td>
<td>3.90</td>
<td>4.65</td>
<td>3.95</td>
<td>44.694</td>
<td>896.30</td>
</tr>
</tbody>
</table>

10000000 iterations
LogSumExp

- LLVM Optimization Passe
- Constant (wrt inputs) detection

```c
#define N 10000000
double logsumexp(double* x, size_t n) {
    double A = 0;
    for(int i=1; i < n; i++) {
        A = max(A, x[i]);
    }
    double sema = 0;
    for(int i=0; i < n; i++) {
        sema += max(x[i] - A);
    }
    return max(sema) + A;
}
```
Taylor Expand Log

- LLVM Optimization Passe
- Constant (wrt inputs) detection

10000000 iterations

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.74</td>
<td>3.72</td>
</tr>
<tr>
<td>Forward</td>
<td>3.74</td>
<td>4.56</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>3.90</td>
<td>4.65</td>
</tr>
</tbody>
</table>
LogSumExp

- LLVM Optimization Passe
- Constant (wrt inputs) detection

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.364</td>
<td>0.364</td>
</tr>
<tr>
<td>Forward</td>
<td>0.364</td>
<td>2.994</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>0.605</td>
<td>3.836</td>
</tr>
</tbody>
</table>

100000000 elements
Gradient Descent:

- LLVM Optimization Passe
- Constant (wrt inputs) detection

Find Matrix by Gradient Descent

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>4.731</td>
<td>25.606</td>
</tr>
<tr>
<td>Gradient Descent</td>
<td>22.672</td>
<td>133.354</td>
</tr>
</tbody>
</table>
Training Simple Neural Network

- LLVM Optimization Passee
- Constant (wrt inputs) detection

<table>
<thead>
<tr>
<th>Enzyme</th>
<th>Adept</th>
<th>Handwritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>73.718</td>
<td>338.097</td>
<td>72.178</td>
</tr>
</tbody>
</table>

Picked first C MNIST Code on Github:

https://github.com/AndrewCarterUK/mnist-neural-network-plain-c

- 1-layer fully connected layer => softmax => cross-entropy loss
- Batch size 100
- 1000 iterations
- Learning rate 0.5
Case Study: Subcall

- Previous parallel IR's based on CFG's model parallel tasks symmetrically.

```c
double loadsq(double* x) {
    return x[0] * x[0];
}
void f(double* x) {
    *x = loadsq(x);
}
void diffe_f(double* x,
               double* xp) {
    __enzyme_autodiff(f, x, xp);
}
```

```c
define double @loadsq(double* %x)
```

```c
entry
%val = load %x
%mul = %val * %val
ret %mul
```

```c
define void @f(double* %x)
```

```c
entry
%call = @loadsq(%x)
store %x = %call
ret
```
Case Study: Read Sum

```c
double loadsq(double* x) {
    return x[0] * x[0];
}
void f(double* x) {
    *x = loadsq(x);
}
```

```c
define {double, double} @augment_loadsq(double* %x)
entry
%val = load %x
%mul = %val * %val
ret {/*return val*/%mul,
    /*cache*/ %val}
```

```c
define void @diffe_loadsq(double* %x, double* %x', double %diffe, double %cache)
entry
%val = %cache // cannot reload as x changed
%mul = %val * %val
%mul' = %diffe
%val' = 2 * %val * %mul'
store %x' += %val'
```
define {double, double} @augment_loadsq(double* %x)

entry

%val = load %x
%mul = %val * %val
ret { /*return val*/ %mul,
    /*cache*/ %val}

define void @diffe_loadsq(double* %x, double* %x', double %diffe, double %cache)

entry

%val = %cache // cannot reload as x changed
%mul = %val * %val
%mul' = %diffe
%val' = 2 * %val * %mul'
store %x' += %val'

define void @diffe_f(double* %x)

entry

{%call, %cache} = @augment_loadsq(%x)
store %x = %call
%call' = load %x'
store %x' = 0
@augment_loadsq(%x, %x', %call', %cache)
ret

double loadsq(double* x) {
    return x[0] * x[0];
}

void f(double* x) {
    *x = loadsq(x);
}
Case Study: Read Sum

\[ \text{load} \%x \]
\[ \text{mul} = \text{load} \%x \times \text{load} \%x \]
\[ \text{return} \{*/\text{return val}*/\text{mul}, */\text{cache}*/ \text{load} \%x\} \]

\[ \text{define void} \ @\text{diffe_loadsq}(\text{double*} \ %x', \ \text{double} \ %diffe, \ \text{double} \ %cache) \]
\[ \text{store} \ %x' += 2 \times \text{cache} \times \text{diffe} \]

\[ \text{define void} \ @\text{diffe_f}(\text{double*} \ %x) \]
\[ \{\text{call}, \text{cache}\} = @\text{augment_loadsq}(\%x) \]
\[ \text{store} \ %x = \text{call} \]
\[ \text{call'} = \text{load} \%x' \]
\[ \text{store} \ %x' = 0 \]
\[ @\text{augment_loadsq}(\%x', \ \text{call'}, \ \text{cache}) \]
\[ \text{ret} \]
Type Analysis: TODO REDO THIS

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
Load + Store Propagation

```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
TBAA Propagation

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```
callee:

void callee(int* ptr) {
    ptr: {}  
    ptr2: [{[]:Pointer, [0]:Double}]  
    loadtype: [{[]:Double}]  
    ptr3: {}  
    cptr2: [{[]:Pointer}]  
    notype: {}  
    cptr3: [{[]:Pointer, [0]:Int}]  
```

ptr2 = indirect
ptr3 = indirect
cptr3 => ptr

```c
int* indirect(int* x, int idx)
{
    return &x[idx];
}

void callee(int* ptr)
{
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int cptr2 = &ptr[2];
    int notype = *cptr2;
    int cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```c
void callee(int* ptr)
{
    ptr2 = indirect(ptr3 = indirect
```
ptr => cptr2

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}
void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```c
void callee(int* ptr) {
    ptr:      {[]:Pointer, [24]:Int}
    ptr2:     {[]:Pointer, [0]:Double}
    loadtype: {[]:Double}
    ptr3:     {}
    cptr2:    {[]:Pointer, [8]:Int}
    notype:   {}
    cptr3:    {[]:Pointer, [0]:Int}
}
ptr2 = indirect
ptr3 = indirect
```
ptr2 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```c
void callee(int* ptr) {
    ptr:      {
        [24]:Pointer,
        [24]:Int
    }
    ptr2:     {
        [0]:Double
    }
    loadtype: {
        [0]:Double
    }
    ptr3:     {
    }
    cptr2:    {
        [8]:Int
    }
    notype:   {
    }
    cptr3:    {
        [0]:Pointer,
        [0]:Int
    }
}
```

ptr2 = indirect

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}
```

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>[24]:Pointer</td>
</tr>
<tr>
<td>ptr2</td>
<td>[0]:Double</td>
</tr>
<tr>
<td>loadtype</td>
<td>[0]:Double</td>
</tr>
<tr>
<td>ptr3</td>
<td></td>
</tr>
<tr>
<td>cptr2</td>
<td>[8]:Int</td>
</tr>
<tr>
<td>notype</td>
<td></td>
</tr>
<tr>
<td>cptr3</td>
<td>[0]:Pointer</td>
</tr>
</tbody>
</table>

x:      {
    [24]:Pointer,
    [24]:Int
}
idx:    {
    [0]:Int@2
}
&x[idx] {} return {
    [0]:Pointer,
    [0]:Double
}
```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
    return;
}
```

```
void callee(int* ptr) {
    ptr: {
        [x]: Pointer, [24]: Int
    }
    ptr2: {
        [x]: Pointer, [0]: Double
    }
    loadtype: {
        [x]: Double
    }
    ptr3: {}
    cptr2: {
        [x]: Pointer, [8]: Int
    }
    notype: {}
    cptr3: {
        [x]: Pointer, [0]: Int
    }
}
```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}

ptr2 = indirect
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
ptr2 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
ptr2 Call IPO

• New interprocedural analysis that detects the type of data

• Perform series of fixed-point updates propagating type information to uses/users

• Each value has a set of memory offsets: type

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
void callee(int* ptr) {
    ptr: {[]:Pointer, [16]:Double, [24]:Int}
    ptr2: {[]:Pointer, [0]:Double, [8]:Int}
    loadtype: {[]:Double}
    ptr3: {}
    cptr2: {[]:Pointer, [8]:Int}
    notype: {}
    cptr3: {[]:Pointer, [0]:Int}
}
```

```c
ptr2 = indirect
```

```c
int* indirect(int* x, int idx) {
    x: {[]:Pointer, [16]:Double, [24]:Int}
    idx: {[]:Int@2}
    &x[idx] {[]:Pointer, [0]:Double, [8]:Int}
    return {[]:Pointer, [0]:Double, [8]:Int}
}
```
ptr => cptr2

- New interprocedural analysis that detects the underlying type of data
- Perform series of fixed-point updates propagating type information to uses/users
- Each value has a set of memory off sets: type

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *double ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t)cptr3) = 100;
}
```
**cptr2 => notype**

- New interprocedural analysis that detects the underlying type of data.

- Perform series of fixed-point updates propagating type information to uses/users.

- Each value has a set of memory offsets: type `int* indirect(int* x, int idx) { return &x[idx]; }

```c
void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

- **callee**:
  ```c
  void callee(int* ptr) {
    ptr:      {[ ]: Pointer, [16]: Double, [24]: Int}
    ptr2:     {[ ]: Pointer, [0]: Double, [8]: Int}
    loadtype: {[ ]: Double}
    ptr3:     {}
    cptr2:    {[ ]: Pointer, [0]: Double, [8]: Int}
    notype:   {[ ]: Double}
    cptr3:    {[ ]: Pointer, [0]: Int}
  ```
ptr3 Call IPO

- New interprocedural analysis that detects the type of data

- Perform series of fixed-point updates propagating type information to uses/users

- Each value has a set of memory offsets:

  ```c
  int* indirect(int* x, int idx) {
    return &x[idx];
  }
  
  void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr*, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
  }
  
  void callee(int* ptr) {
    ptr: {[]:Pointer, [16]:Double, [24]:Int}
    ptr2: {[]:Pointer, [0]:Double, [8]:Int}
    loadtype: {[]:Double}
    ptr3: {}
    cptr2: {[]:Pointer, [0]:Double, [8]:Int}
    notype: {[]:Double}
    cptr3: {[]:Pointer, [0]:Int}
  }
  
  ptr3 = indirect
  ```
ptr3 Call IPO - x

- New interprocedural analysis that detects the underlying type of data
- Perform series of fixed-point updates propagating type information to uses/users
- Each value has a set of memory offsets and types:

```
int* indirect(int* x, int idx) {
    return &x[idx];
}
```

```
void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
ptr3 Call IPO - return

- New interprocedural analysis that detects the type of data
- Perform series of fixed-point updates propagating type information to uses/users
- Each value has a set of memory offsets: type

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
ptr3 = indirect
```
ptr3 Call IPO

- New interprocedural analysis that detects the underlying type of data
- Perform series of fixed-point updates propagating type information to uses/users
- Each value has a set of memory offsets: type int*

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
void callee(int* ptr) {
    ptr: {[]:Pointer, [16]:Double, [24]:Int}
    ptr2: {[]:Pointer, [0]:Double, [8]:Int}
    loadtype: {[]:Double}
    ptr3: {[]:Pointer, [0]:Int}
    cptr2: {[]:Pointer, [0]:Double, [8]:Int}
    notype: {[]:Double}
    cptr3: {[]:Pointer, [0]:Int}
}
```
• New interprocedural analysis that detects the underlying type of data
• Perform series of fixed-point updates propagating type information to uses/users
• Each value has a set of memory offsets: type

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
LLVM IR

LLVM represents each function as a **control-flow graph (CFG)** of **BasicBlocks**, containing lists of **Instructions**.

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

---

**Control flow**

- **Entry**: %cmp = %n < 0
- **Branch**: br %cmp, exit, if.else

**Basic Blocks**

- **If.else**: %nm1 = %n - 1
  - %x = fib(%nm1)
  - %nm2 = %n - 1
  - %y = fib(%nm2)
  - %add = %x + %y
  - br exit

- **Exit**: rv = phi([n, entry], [add, if.else])
  - return rv