Enzyme: High-Performance Automatic Differentiation of LLVM

William S. Moses
Valentin Churavy

wmoses@mit.edu
LAFI '21
Jan 17, 2020
Differentiation Is Key To Machine Learning

• 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 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
  
  - 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

- C++ → AD → C++
- Julia → AD → Julia
- R → AD → R
- AD → Lower → LLVM → AD
- AD → Optimize → AD
- AD → CodeGen → EXE
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

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

// Compute norm in O(n)
void norm(double[] out, double[] in) {
    double res = mag(in);
    for (int i=0; i<n; i++) {
        out[i] = in[i] / res;
    }
}
```
Optimization & Automatic Differentiation

\[ O(n^2) \]

\[
\begin{align*}
\text{for } i=0..n \{ \\
\text{out}[i] &\text{ /= mag(in)} \\
\}
\end{align*}
\]

\[ O(n) \]

\[
\begin{align*}
\text{res} &= \text{mag(in)} \\
\text{for } i=0..n \{ \\
\text{out}[i] &\text{ /= res} \\
\}
\end{align*}
\]

\[ O(n) \]

\[
\begin{align*}
\text{d_res} &= 0.0 \\
\text{for } i=n..0 \{ \\
\text{d_res} &\text{ += d_out[i]} \\
\} \\
\n\text{\nabla mag}(d\text{\_in}, d\text{\_res})
\end{align*}
\]
Optimization & Automatic Differentiation

\[ O(n^2) \]

for \( i = 0 \ldots n \) {
    out[\( i \)] /= \( \text{mag}(\text{in}) \)
}

Optimize

\[ O(n) \]

res = \( \text{mag}(\text{in}) \)
for \( i = 0 \ldots n \) {
    out[\( i \)] /= res
}

AD

\[ O(n) \]

d_res = 0.0
for \( i = n \ldots 0 \) {
    d_res += d_out[\( i \)]
}
\[ \nabla \text{mag}(\text{d_in}, \text{d_res}) \]

\[ O(n^2) \]

for \( i = 0 \ldots n \) {
    out[\( i \)] /= \( \text{mag}(\text{in}) \)
}

AD

\[ O(n^2) \]

for \( i = n \ldots 0 \) {
    d_res = d_out[\( i \)]
\[ \nabla \text{mag}(\text{d_in}, \text{d_res}) \]
Optimization & Automatic Differentiation

\[ \mathcal{O}(n^2) \]

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

\[ \mathcal{O}(n) \]

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

\[ \mathcal{O}(n^2) \]

```c
d_res = 0.0
for i=n..0 {
    d_res += d_out[i]...
}
\n\langle \text{mag}(\text{d_in}, \text{d_res}) \rangle
```

\[ \mathcal{O}(n) \]

```c
for i=n..0 {
    d_res = d_out[i]...
    \langle \text{mag}(\text{d_in}, \text{d_res}) \rangle
}
```

\[ \mathcal{O}(n^2) \]
Optimization & Automatic Differentiation

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

\[
O(n^2) \quad \rightarrow \quad O(n) \quad \rightarrow \quad O(n)
\]

Optimize

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

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

AD

\[
d_{\text{res}} = 0.0 \\
\text{for } i=n..0 \{ \\
\quad d_{\text{res}} += d_{\text{out}[i]}... \\
\}
\]

\[
\nabla \text{mag}(d_{\text{in}}, d_{\text{res}})
\]

\[
O(n^2) \quad \rightarrow \quad O(n^2) \quad \rightarrow \quad O(n^2)
\]

AD

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

\[
\text{for } i=n..0 \{ \\
\quad d_{\text{res}} = d_{\text{out}[i]}... \\
\quad \nabla \text{mag}(d_{\text{in}}, d_{\text{res}}) \\
\}
\]

Optimize
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”
- Well-defined semantics
- Large collection of optimizations and analyses
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

define double @relu3(double %x) {

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

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

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

Case Study: ReLU3

Active Instructions

```mlang
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
```
Define a function `@diffe_relu3(double %x, double %differet)`

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

Allocate and zero shadow memory for active values

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

Entry

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

Cond.true

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

Cond.end

Allocate & zero shadow memory for active values
```
define double @diffe_relu3(double %x, double %differ)

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

%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

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

Compute adjoints for active instructions
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

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

alloca %x'
alloca %call'
alloca %result'

%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

%0 = load %x'
ret %0
Essentially the optimal hand-written gradient!

define double @diffe_relu3(double %x)

```cpp
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

```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];
    src'[1] += dst'[1];
    dst'[0] = 0;
    dst'[1] = 0;
}
```
Challenges of Low-Level AD

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

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

- Collection of benchmarks from Microsoft’s ADBench suite and of technically interest
Speedup of Enzyme

Enzyme is $4.2x$ faster than Reference!
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
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow
- Open source (enzyme.mit.edu & join our mailing list)
- Current work: GPU AD, MPI AD
Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Hal Finkel, Laurent Hascoet, Paul Hovland, Jan Hueckelheim, 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

- 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
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow
- Open source (enzyme.mit.edu & join our mailing list)
- Current work: GPU AD, MPI AD
END
**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

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);

// diffeDupnoneed 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);
}
```
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]

Differentiation Is Key To Machine Learning

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

// C++ nbody simulator

```cpp
#include <vector>

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

```python
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
```
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
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)