Hi all,

Automatic differentiation (AD) is a key component in algorithms used in machine learning, scientific computing, and elsewhere.

For the last year-and-a-half, the `Enzyme` group have been looking at the practical possibility of doing automatic differentiation as part of the LLVM optimization pipeline. Performing automatic differentiation in LLVM is quite beneficial as it allows all of the languages that lower to LLVM to incorporate automatic differentiation without much additional work. It also allows for automatic differentiation across languages, which is similarly beneficial.

One unexpected benefit we found of doing AD at the LLVM-level is that there is a significant performance benefit (4.2x in our tests) to be gained by performing AD after LLVM's optimization passes [1].

After several months of testing with various users including the Rust [4, 5], C/C++, Julia [6], Fortran, and machine learning communities, we'd like to share LLVM-based automatic differentiation more widely and ask to be considered as an LLVM incubator project.

Our code is available here (https://github.com/wsmoses/Enzyme/tree/master/enzyme) as a plugin for LLVM versions 7 through master. We've had weekly meetings for the past several months with folks from MIT, Argonne, Princeton, Google, NVIDIA, and Facebook and welcome anyone who wants to join. Documentation and install instructions for Enzyme is available here: https://enzyme.mit.edu. We have our charter available here:
https://docs.google.com/document/d/10IK2EgZa-4WF0IOSIkND1_cX3IQLAxEVSOWqbQzNpcs/edit#

Performing automatic differentiation inside of LLVM presents several interesting technical questions, which we've explored with the community in a poster and SRC talk at the 2020 US LLVM Dev Meeting [2, 3].

The Enzyme team

[1]
https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf
[2] https://c.wsmoses.com/posters/Enzyme-llvmdev.pdf
[3] https://www.youtube.com/watch?v=auQNFDIaXdM,
https://c.wsmoses.com/presentations/enzyme-llvmdev-reduced.pdf
[4] https://github.com/tiberiusferreira/oxide-enzyme https://github.com/bytesnake/oxide-enzyme,
[5]
https://internals.rust-lang.org/t/automatic-differentiation-differential-programming-via-llvm/13188
[6] https://github.com/wsmoses/Enzyme.jl

Charter
---
title: "Charter"
date: 2020-01-12T16:00:15Z
draft: true
weight: 10

Enzyme is a project for high-performance automatic differentiation. The Enzyme community aims to be open and welcoming. If you'd like to participate, you can do so in a number of ways.

 * Join our [mailing list](https://groups.google.com/d/forum/enzyme-dev)
 * Join our weekly open-design call (see mailing list for meeting link)
 * Participate in development on [Github](https://github.com/wsmoses/Enzyme)

# Abstract

Applying differentiable programming techniques and machine learning algorithms to foreign programs requires developers to either rewrite their code in a machine learning framework, or otherwise provide derivatives of the foreign code. This project develops Enzyme, a high-performance automatic differentiation (AD) compiler plugin for the LLVM compiler framework capable of synthesizing gradients of statically analyzable programs expressed in the LLVM intermediate representation (IR). Enzyme synthesizes gradients for programs written in any language whose compiler targets LLVM IR including C, C++, Fortran, Julia, Rust, Swift, MLIR, etc., thereby providing native AD capabilities in these languages. Unlike traditional source-to-source and operator-overloading tools, Enzyme performs AD on optimized IR.

# Introduction

Machine learning (ML) frameworks such as PyTorch and TensorFlow have become widespread as the primary workhorses of the modern ML community. Computing gradients necessary for algorithms such as backpropagation, Bayesian inference, uncertainty quantification, and probabilistic programming requires all of the code being differentiated to be written in these frameworks. This is problematic for applying ML to new domains as existing tools like physics simulators, game engines, and climate models are not written in the domain specific languages (DSL's) of ML frameworks. The rewriting required has been identified as the quintessential challenge of applying ML to scientific computing. As stated by Rackauckas "this is [the key challenge of scientific ML] because, if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train."

To remedy this issue, the trend has been to either create new DSL's that make the rewriting process easier or to add differentiation as a first-class construct in programming languages. This results in efficient gradients, but still requires rewriting in either the DSL or the differentiable programming language. Developers may want to use code foreign to a ML framework to either re-use existing tools or write loss functions in a language with an easier abstraction for their use case. While there exist reverse-mode automatic differentiation (AD) frameworks for various languages, using them automatically on foreign code for an ML framework is difficult as they still require rewriting and have limited support for cross-language AD and libraries. The two primary approaches to computing gradients are as follows.

- ***Operator-overloading*** tools compute derivatives by providing differentiable versions of existing language constructs. Examples include Adept/ADOL-C, C++ libraries providing differentiable types; and JAX/Autograd, Python libraries providing derivatives of NumPy-style functions. These approaches, however, require rewriting programs to use differentiable operators in place of standard language utilities. This prevents differentiation of many libraries and code in other languages.
- ***Source-rewriting*** tools analyze the source code of programs and emits source code defining the gradient. Examples of tools include Tapenade for C and Fortran; ADIC for C and C++; and Zygote for Julia. Users must provide all code being differentiated to the tool ahead-of-time and must write programs in a specific subset of the language. This makes source-rewriting hard to use with header-only libraries and impossible to use with precompiled libraries. Both operator-overloading and source-rewriting AD systems differentiate programs before optimization.

Performing AD on unoptimized programs, however, may result in complicated gradients that cannot be simplified by future optimization.

For example, consider the following program that normalizes a vector in O(N^2) time. Running loop-invariant code-motion (LICM) reduces the runtime to O(N) by moving the call to `mag` outside the loop.

```
// Compute magnitude in O(N)
float mag(const float* x);

void norm(float* out, float* in) {
    // code motion optimization can move outside the loop
    // float res = mag(in);
    for(int i=0; i<N; i++) {
      out[i] = in[i]/mag(in);
    }
}

void __enzyme_autodiff(void*, ...);

void grad_norm(float* out, float* d_out, float* in, float* d_in) {
  __enzyme_autodiff((void*)norm, out, d_out, in, d_in);
}
```

Differentiating the O(N) optimized program results in the O(N) gradient on the left, which has the corresponding grad_mag call outside the loop. If AD is run first, then the call to `grad_mag` remains inside the loop as shown on the right. A subsequent run of the LICM optimization, however, cannot move the call to `grad_mag` outside the loop as it uses the variable `d_res`, defined in the loop.

Loop-Invariant Code Motion, then AD, O(N)

```
void grad_norm(float* out, float* d_out,
               float*  in, float* d_in)
{
  float res = mag(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i]/res;
  }
  float d_res = 0;
  for (int i=0; i<N; i++) {
    d_res += -in[i]*in[i]/res \
                    * d_out[i];
    d_in[i] += d_out[i]/res;
  }
  grad_mag(in, d_in, d_res);
}
```

AD, then LICM, O(N^2)

```
void grad_norm(float* out, float* d_out,
               float*  in, float* d_in)
{
  float res = mag(in);
  for (int i=0; i<N; i++) {
    out[i] = in[i]/res;
  }
  for (int i=0; i<N; i++) {
    float d_res = -in[i]*in[i]/res \
                          * d_out[i];
    d_in[i] += d_out[i]/res;
    grad_mag(in, d_in, d_res);
  }
}
```

Traditional AD systems have not operated on optimized intermediate representation (IR) as doing so requires either re-implementing all of the optimizations or working at a low-level after which optimization has already been performed. We remedy this by implementing Enzyme inside of LLVM directly.

In addition to allowing better performance by integration with LLVM's existing optimization and analysis passes, this also allows AD to be implemented once for a variety of languages. Moreover, AD within LLVM allows for cross-language AD. Enzyme can also use existing pieces of LLVM infrastructure such as link-time optimization and embedded bitcode to enable multisource AD.

# Type Analysis

Performing automatic differentiation at a low-level presents additional challenges as several pieces of high-level information that AD traditionally relies upon may be absent. Consider the following program which copies 8 bytes of data.

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

Depending on what the underlying type of the data is, a different reverse pass is required.

Gradient memcpy for double inputs	Gradient memcpy for float input
`void grad_f(double* dst, double* ddst,`	`void grad_f(float* dst, float* ddst,`
`            double* src, double* dsrc) {`	`            float* src, float* dsrc) {`
`  // Forward pass`	`  // Forward pass`
`  memcpy(dst, src, 8);`	`  memcpy(dst, src, 8);`
`  // Reverse pass`	`  // Reverse pass`
`  dsrc[0] += ddst[0];`	`  dsrc[0] += ddst[0];`
`  ddst[0] = 0;`	`  ddst[0] = 0;`
	`  dsrc[1] += ddst[1];`
	`  ddst[1] = 0;`
`}`	`}`

Enzyme creates a new interprocedural fixedpoint analysis rather than relying on types prescribed by LLVM. Every value in a function is given a type tree that describes the known type at any given byte offset in the value. If the type at a particular offset is a pointer type, we have a new type tree that represents the types inside that offset.

# Activity Analysis

Activity analysis determines what instructions could impact the gradient computation and is common in automatic differentiation systems to avoid performing unnecessary adjoints. Enzyme also uses activity analysis to avoid taking gradients of “undifferentiable” instructions such as the cpuid instruction. An instruction is active if and only if it can propagate a differential value to its return or another memory location. For example, a function that counts the length of a an active input array would not be active. In our implementation of activity analysis, we leverage LLVM’s alias analysis and type analysis to help prove that instructions are inactive. As an example, any read-only function that returns an integer must be inactive since it cannot propagate differential values through the return or any memory location. This is true because the differential value of any integer value must be zero and while the instruction can read active memory it cannot propagate it anywhere.

# Parallelism

Support for parallelism is ongoing. Performing automatic differentiation on parallel code adds additional complexities around races. A benign read-race in the forward-pass becomes a write-races in the reverse pass which can lead to incorrect code. In addition to integrating with existing LLVM tools for automatically parallelizing code, Enzyme currently supports a limited subset of OpenMP and CUDA codes as input, using atomic operations to ensure correctness. A current research effort is improving performance through the use of parallel reductions and expanding the scope of parallel programs handled as inputs.