Enzyme: High-Performance Automatic Differentiation of LLVM

William S. Moses (wmoses@mit.edu), Valentin Churavy (vchuravy@mit.edu)

MIT CSAIL

Existing Automatic Differentiation Tools

Computing derivatives is key to many machine learning algorithms. Existing approaches:

- **Differentiable DSLs** (TensorFlow [1], PyTorch [2], DiffTaichi [3]) provide a new language where everything is differentiable. Must rewrite code.
- **Operator Overloading** (Adept [4], JAX [5]) tools provide differentiable versions of existing language constructs. May require rewriting.
- **Source Rewriting** tools statically analyze code to produce a new gradient function in the source language.

This hinders application of ML to new domains!

```
// Pseudo-Relativistic C++ nbody simulator
vec3 force(Planet& p1, Planet& p2) {
  // Approximate Einstein Field Equation
  // G_{μν} + Λg_{μν} = (8πG/c^4) T_{μν}
}
void step(std::array<Planet> bodies, double dt) {
  for (size_t i=0; i<bodies.size(); i++) {
   for (size_t j=0; j<bodies.size(); j++) {
     if (i == j) continue;
     bodies[i].acc += force(bodies[i], bodies[j]) * dt;
   }
  }
  ...
}
```

Figure 1. Currently, ML researchers who want to use existing libraries like the relativistic simulator above, must spend their time fully understanding and rewriting the implementation of the library rather than using it to solve their problem.

Optimization and AD

All tools for existing code operate at the source level preventing optimizations before AD without reimplementing compiler analyzes and optimization into the AD tool. While historically not considered necessary, we demonstrate in Figure 2 how crucial optimization prior to AD can be.

```
float mag(const float*); //Compute magnitude in O(N)
void norm(float* out, const float* in){
  // float res = mag(in); LICM moves mag outside loop
  for(int i = 0; i < N; i++) { out[i] = in[i] / mag(in); }
}
```

```
// LICM, then AD, O(N)
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]
           * d_out[i]/res;
  d_in[i] += d_out[i]/res;

}
∇mag(in, d_in, d_res);
```

```
// AD then LICM, O(N^2)
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]
              * d_out[i]/res;
  d_in[i] += d_out[i]/res;
  ∇mag(in, d_in, d_res);
}
//
```

Figure 2. When differentiating **norm**, running LICM prior to AD is asymptotically faster than running AD followed by LICM.

Usage

We provide Enzyme packages for PyTorch and TensorFlow that allow users to import foreign code into their ML workflow without rewriting.

```
import torch
import enzyme
# Create some initial tensor
inputTensor = ...
# Apply foreign function
outputTensor = enzyme("test.c", "f").apply(inputTensor)
# Derive gradient
outputTensor.backward()
print(inputTensor.grad)
```

Figure 3. Using Enzyme from PyTorch.

Design

Conventional Wisdom: "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 [6]

Enzyme overturns said wisdom by demonstrating successful and high-performance AD on low-level programs. By introducing new interprocedural analyses, Enzyme is able to extract all the required high-level semantics necessary to differentiate.

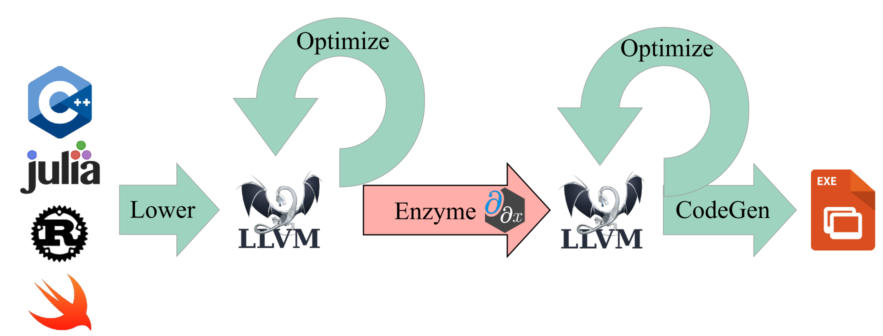

Figure 4. Enzyme differentiates LLVM [7] intermediate representation. This allows Enzyme to differentiate a variety of languages and act both before and after optimization.

```
define @relu3(double %x)
entry:
  ; Shadows for reverse
  ; alloca %d_x = 0.0
  ; alloca %d_call = 0.0
  ; alloca %d_result = 0.0
  ; Cache of %cmp
  ; alloca %cmp_cache
  %cmp = %x > 0
  br %cmp, %iftrue, %end
iftrue:
  %call = @pow(%x, 3)
  br cond.end
end:
  %res = φ[%call, if.true],
   ↪ [0, entry]
  ret %res
```

```
rev_end:
  ; adjoint of return
  store %d_res = 1.0
  ; adjoint of %res phi node
  %cmp2 = load %cmp_cache
  %tmp = load %d_res
  %d_call += if %cmp2, %tmp else 0
  store %d_res = 0.0
  br %cmp, %rev_iftrue, %rev_entry
rev_iftrue:
  ; adjoint of %call
  %df = 3 * @pow(%x, 2)
  %d_x += %df * (load %d_call)
  store %d_call = 0.0
  br %rev_entry
rev_entry:
  %0 = load %d_x
  ret %0
```

Figure 5. Gradient synthesis of `relu(pow(x,3))`. *Left:* the original computation with comments showing the shadow allocations of active variables that would be added to the forward pass. *Right:* reverse pass generated by Enzyme. The full synthesized gradient function would combine these (with shadow allocations added), replacing the return with a branch to the reverse pass.

Evaluation

Performing AD after optimization yields a $4.2\times$ speedup over AD before optimization. This accounts for much, but not all, of Enzyme's improvement over prior art (different cache and activity analysis implementations).

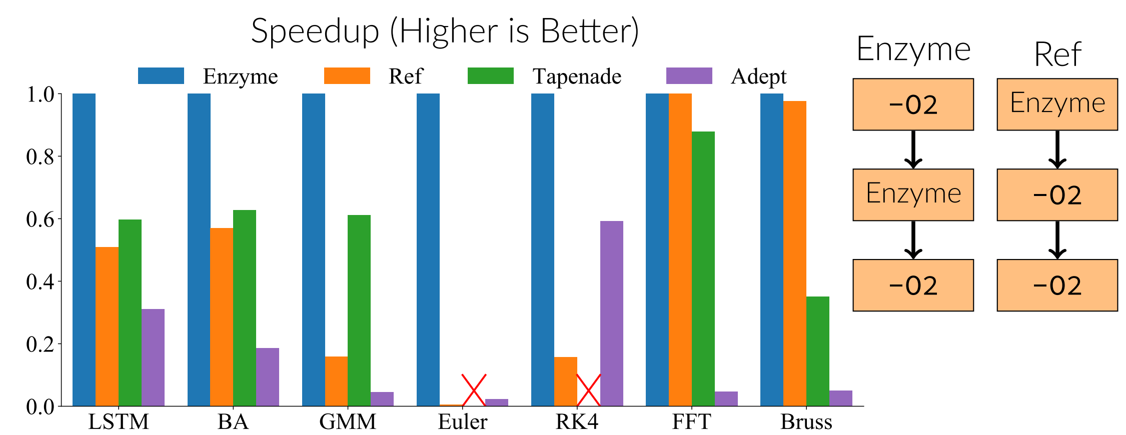

Figure 6. Relative speedup of AD systems on ADBench+ [8] benchmarks, higher is better. A red X denotes programs that an AD system does not produce a correct gradient. A value of 1.0 denotes the fastest system, whereas 0.5 denotes taking twice as long.

Conclusion

Enzyme is compiler plugin that performs reverse-mode automatic differentiation of LLVM [7]. By performing AD after optimization, Enzyme is able to achieve state-of-the-art performance. It is easy to incorporate into existing tools and we have demonstrated taking derivatives of C/C++ via Clang, PyTorch [2], and Tensorflow [1]; as well as Julia, Rust, and Swift. We've also demonstrated dynamic language support by using Enzyme to differentiate Julia [9].

For more information about installing and using Enzyme, please visit `https://enzyme.mit.edu` and come to our spotlight presentation!

References & Acknowledgements

[1] M. Abadi *et al.*, TensorFlow: A system for large-scale machine learning, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[2] A. Paszke *et al.*, Automatic differentiation in PyTorch, in *NIPS 2017 Workshop Autodiff*, 2017.

[3] Y. Hu *et al.*, arXiv preprint arXiv:1910.00935 (2019).

[4] R. J. Hogan, ACM Transactions on Mathematical Software (TOMS) 40, 1 (2014).

[5] J. Bradbury *et al.*, JAX: composable transformations of Python+NumPy programs, 2018.

[6] M. Innes, arXiv preprint arXiv:1810.07951 (2018).

[7] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[8] F. Srajer, Z. Kukelova, and A. Fitzgibbon, Optimization Methods and Software 33, 889 (2018).

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM review 59, 65 (2017).

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323. Valentin Churavy was supported in part by the DARPA program PAPPA under Contract Number HR00112090016, 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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.