Enzyme: High-Performance Automatic Differentiation of LLVM

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

MIT CSAIL

Existing Automatic Differentiation Tools

- **Differentiable DSLs** (TensorFlow, PyTorch, DiffTaichi) provide a new language where everything is differentiable. Must rewrite code in DSL.
- **Operator Overloading** (Adept, JAX) tools provide differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum). May require rewriting to use non-standard utilities for support.
- **Source Rewriting** tools statically analyze code to produce a new gradient function in the source language. Requires all code available ahead of time and is difficult to use with external libraries.

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

Challenges of AD on Low-Level Code

Performing AD on low-level IR presents additional challenges as source-level information is lost. For example, differentiating memcpy in Figure 2 requires the underlying type of the data to select the correct gradient.

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

```
// Assume double inputs
∇f(double* dst, double* ddst,
   double* src, double* dsrc) {
  // Forward pass
  memcpy(dst, src, 8);
  // Reverse pass
  dsrc[0] += ddst[0];
  ddst[0] = 0;


}
```

```
// Assume float inputs
∇f(float* dst, float* ddst,
   float* src, float* dsrc) {
  // Forward pass
  memcpy(dst, src, 8);
  // Reverse pass
  dsrc[0] += ddst[0];
  ddst[0] = 0;
  dsrc[1] += ddst[1];
  ddst[1] = 0;
}
```

Figure 2. ***Top:*** Call to **memcpy** for an unknown 8-byte object. ***Left:*** Gradient for a **memcpy** of 8 bytes of double data. ***Right:*** Gradient for a **memcpy** of 8 bytes of float data.

Design

Enzyme synthesizes derivatives by:

- Running Type and Activity Analysis
- Allocating shadows of active variables
- Creating a "reverse" copy of BasicBlock's in the original code that compute the adjoints of its instructions in reverse order.

```
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 3. 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. ***Bottom:*** demonstration of how to call Enzyme.

- **Type Analysis**: A new interprocedural analysis that derives the underlying types of data by manipulating "TypeTree's" of LLVM Values. TypeTrees are initialized with Constant, TBAA info. Each instruction is given a type propagation rule that until fixpoint. We provide compile-time error if a necessary type cannot be deduced statically.
- **Activity Analysis** determines what instructions could impact derivative computation to avoid computing unnecessary adjoints. Build off Type & Alias Analysis to get better results. 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** is used to store the derivatives of values. Shadow versions of data structures created inside the differentiated function are created automatically. Data structures passed as arguments to the differentiated function must also have shadow arguments passed.
- **Cache**: Some adjoint instructions require values from the forward pass (e.g. $\nabla(x * y) \longrightarrow x * dy + y * dx$). Memory is automatically allocated for all such values. Enzyme optimizes the cache by recomputing instead of caching and avoids caching unnecessary or equivalent values, whenever possible.
- **Extensibility:** Custom gradients are supported by attaching metadata specifying the corresponding gradient function. Multisource AD is supporting by leveraging LLVM's LTO support and libraries with embedded bitcode. This ensures that any potentially active call has a definition available to differentiate.

Evaluation

Performing AD after optimization yields a $4.5\times$ 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).

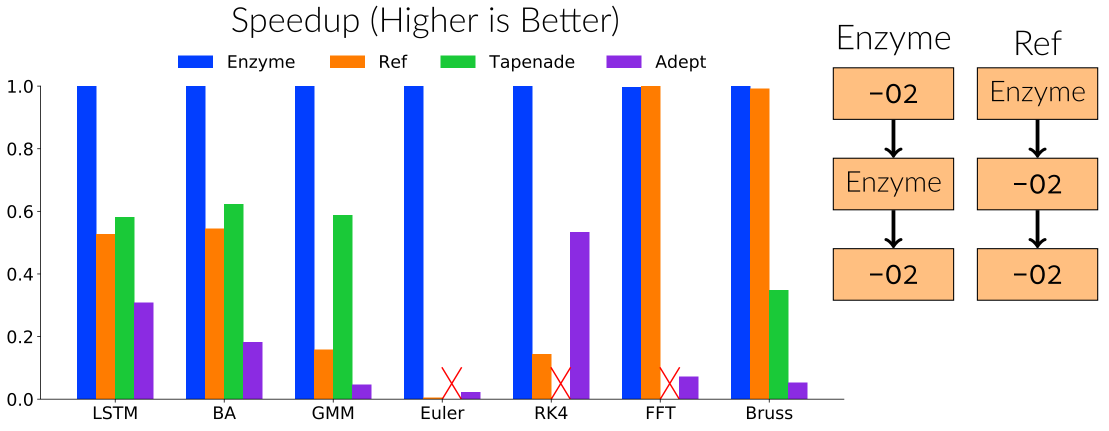

Figure 4. Relative speedup of AD systems on ADBench+ benchmarks, higher is better. A red X denotes programs that an AD system does not produce a correct gradient. A value of 1.0 denotes the fastest system, whereas 0.5 denotes taking twice as long.

Usage

A user can use gradient functions by calling `__enzyme_autodiff` with the function to be differentiated as the first argument. When the Enzyme optimization pass is run, it will replace any calls to `__enzyme_autodiff` with a call a newly-generated gradient function.

```
%grad = call double @__enzyme_autodiff(@relu3, double %x)
        opt -load=/path/to/LLVMEnzyme.so -enzyme
%grad = call double @grad_relu3(double %x)
```

Figure 5. Convention for invoking Enzyme.

Enzyme is built as an LLVM compiler plugin for versions 7 and later ease incorporation into an existing tools. We have demonstrated taking derivatives of C/C++ via Clang, PyTorch, and Tensorflow. We've also demonstrated dynamic language support by using Enzyme to differentiate Julia.

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

Acknowledgements

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.