Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme

William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hückelheim †, Sri Hari Krishna Narayanan †, Michel Schanen †, Johannes Doerfert†

MIT CSAIL*
TUM§, ANL†

Existing Automatic Differentiation Tools

- **Differentiable DSLs** (TensorFlow [1], PyTorch [2]) provide a new language where everything is differentiable. Must rewrite code, only fast if DSL matches abstractions of program.
- **Operator Overloading** (Adept [3], JAX [4]) tools provide differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum). Often creates instruction tape to be interpreted dynamically.
- **Source Rewriting** (Tapenade [5]) tools statically analyze code to produce a new gradient function in the source language. Re-implements parsing/semantics => limited support for recent/complex features.

Optimization and AD

Regardless of approach, all existing AD tools apply on unoptimized source code. Fig 1 demonstrates how applying optimization prior to AD can be asymptotically faster than current approaches.

```
  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);
```

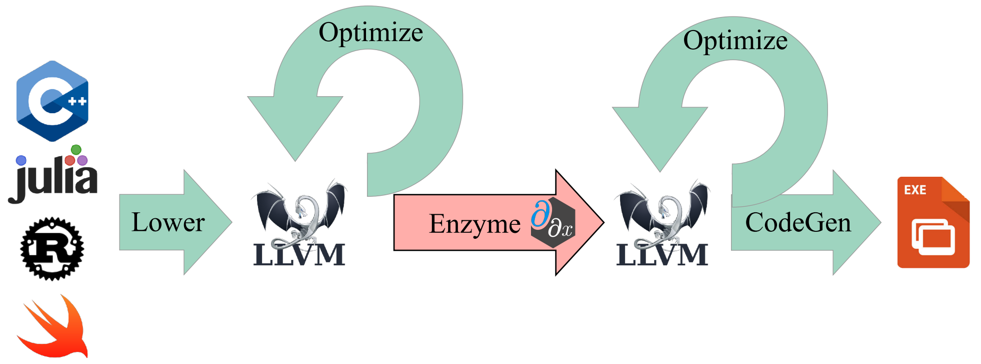
```
// AD then LICM, O(N^2)
float res = mag(in);
for(int i = 0; i < N; i++) {
  out[i] = in[i] / res;
}

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.

Design

Enzyme operates on a common compiler intermediate representation, LLVM [6]. This not only enables Enzyme to operate after optimization (getting speedups like above), but also differentiate any LLVM-based language (C, C++, Fortran, Rust, Swift, Julia, Python, JaX, MLIR, PyTorch, etc).



GPU AD Challenges

Prior work did not explore reverse mode AD of existing GPU code because:

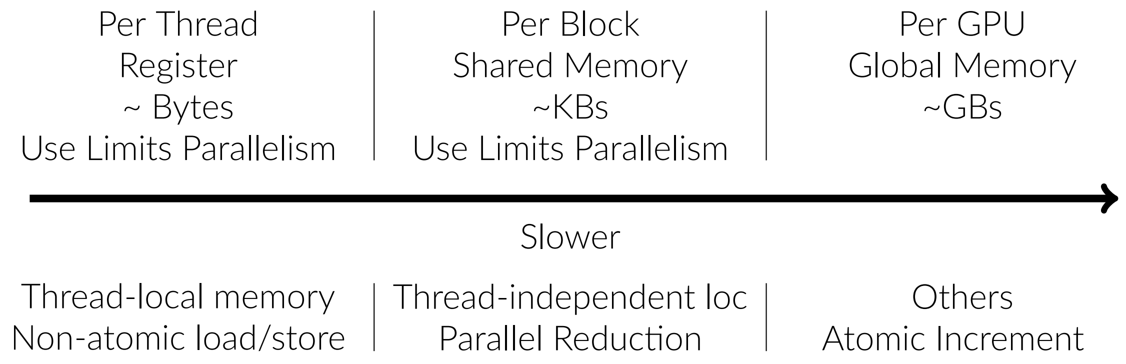
1. Reversing parallel control flow can lead to incorrect results
2. Complex performance characteristics => difficult to write efficient code
3. Resource limitations can prevent kernels from running at all

```
void set(float* ar,
         float val) {


 par_for(i=0; i<10; i++) {
   // Read race
   ar[i] = val;
 }
}
```

```
float ∇set(float* ar, float* d_ar,
           float val) {
  float d_val = 0.0;
  ...
  par_for(i=0; i<10; i++) {
    // ↯ Write race ↯
    d_val += d_ar[i];
    d_ar[i] = 0.0;
  }
  return d_val;
}
```

Figure 2. Differentiating a concurrent read becomes a write-race in the derivative.



Per Thread Register ~ Bytes Use Limits Parallelism	Per Block Shared Memory ~KBs Use Limits Parallelism	Per GPU Global Memory ~GBs

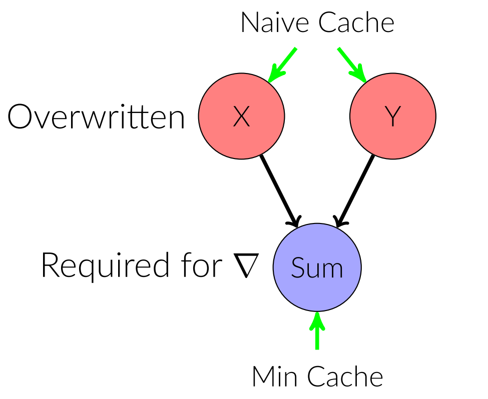
Slower

Thread-local memory Non-atomic load/store	Thread-independent loc Parallel Reduction	Others Atomic Increment

Figure 3. GPU Memory Hierarchy and corresponding AD race resolution mechanism.

GPU & AD-specific Optimizations

Enzyme may need to cache values in order to compute the gradient. Storing too many values in GPU memory can slow down the program dramatically, or even cause it to fail to run. While like on the CPU, existing optimizations reduce overhead, they aren't sufficient. Novel GPU and AD-specific optimizations can speedup by several orders of magnitude.



```
for(int i=0; i<10; i++) {
  float sum = x[i] + y[i];
  use(sum);
}
overwrite(x, y);
grad_overwrite(x, y);
for(int i=9; i>=0; i--) {
  sum = ...
  grad_use(sum);
}
```

Figure 4. Min-Cut Cache Optimization. The variable **sum** is required to compute the derivative, however both of its operands **x** and **y** are overwritten. Rather than caching **x** and **y** to recompute **sum** in the derivative, one can simply cache **sum**. This generalizes to taking the minimum cut of the data-flow graph.

Evaluation

On the CPU (Fig 5), AD after optimization is $4.2\times$ faster than AD before optimization. On the GPU (Fig 6), AD after optimization (blue) is required for most benchmarks to run and novel AD and GPU-specific optimizations (green) provide order-of-magnitude performance improvements.

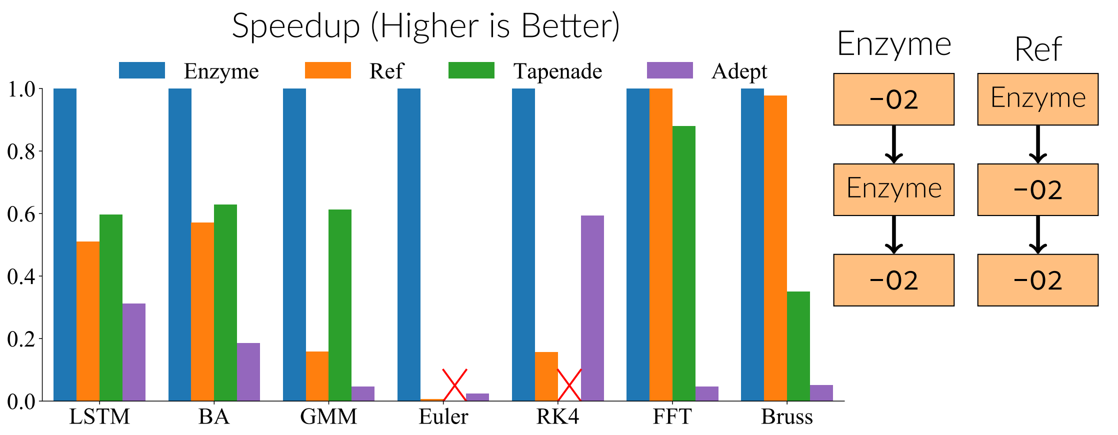


Figure 5. Relative speedup of AD systems on ADBench+ [7] benchmarks, higher is better. A red X denotes programs that an AD system does not produce a correct gradient.

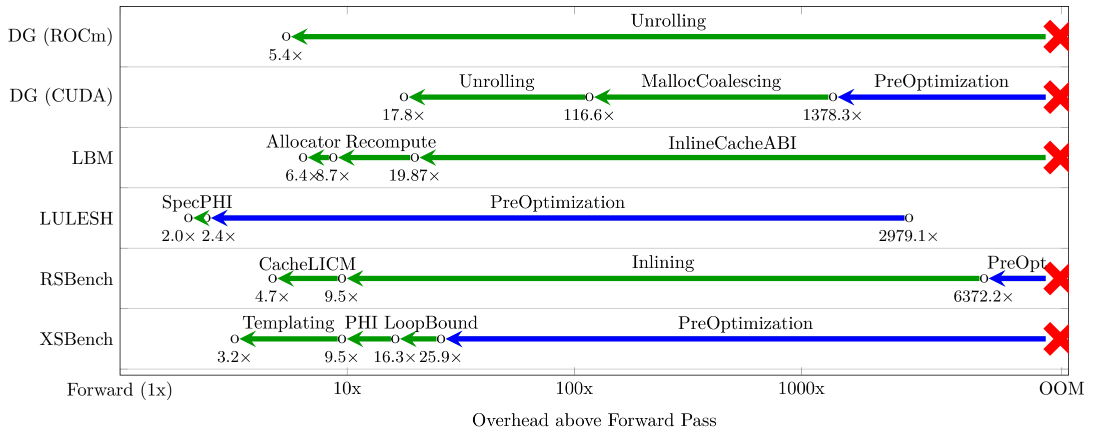


Figure 6. GPU + AD overhead with no optimizations, only LLVM optimizations (blue), and AD-specific optimizations (green). An overhead of N means computing the derivative of all inputs and original outputs is equivalent to running the original code N times.

References & Acknowledgements

For information on installing and using Enzyme, visit `enzyme.mit.edu`.

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

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

[5] L. Hascoët and V. Pascual, ACM Transactions On Mathematical Software 39 (2013).

[6] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis & transformation, in *CGO*, pp. 75–86, IEEE, 2004.

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

This research supported in part by a DOE CSGF DE-SC0019323, in part by the DARPA under Agreement No. HR0011-20-9-0016; in part by NSF Grant OAC-1835443; in part by the German Research Council (DFG) under grant agreement No. 326472365; in part by LANL grant 531711; in part by the DOE under contract number DE-AC02-06CH11357; in part by the Exascale Computing Project (17-SC-20-SC); and in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator under 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.

.