Enzyme: High-Performance Automatic Differentiation

William S. Moses  Valentin Churavy

wmoses@mit.edu
CESMIX Group
May 18, 2021
Differentiation Is Key To Machine Learning And Science

- 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 designed to be differentiated
  • 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

- **Optimize**
- **Lower**
- **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

\[ O(n^2) \]

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

Optimize

\[ O(n) \]

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

\[ \nabla \text{mag}(\text{d_in}, \text{d_res}) \]

\[ O(n) \]

\[
\text{d_res} = 0.0 \\
\text{for } i=n..0 \{ \\
\quad \text{d_res} += \text{d_out}[i] \} \\
\n\text{mag}(\text{d_in}, \text{d_res})
\]
Optimization & Automatic Differentiation

\[ O(n^2) \]

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

\[ O(n) \]

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

\[ O(n) \]

```
d_res = 0.0
for i=n..0 {
    d_res += d_out[i]...
}
∇mag(d_in, d_res)
```

\[ O(n^2) \]

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

\[ O(n^2) \]

```
for i=n..0 {
    d_res = d_out[i]...
    ∇mag(d_in, d_res)
}
```
Optimization & Automatic Differentiation

### $O(n^2)$

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

### $O(n)$

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

### $O(n)$

```
d_res = 0.0
for i=n..0 {
    d_res += d_out[i]...
}
∇mag(d_in, d_res)
```

### $O(n^2)$

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

### $O(n^2)$

```
for i=n..0 {
    d_res = d_out[i]...
    ∇mag(d_in, d_res)
}
```

### $O(n^2)$

```
for i=n..0 {
    d_res = d_out[i]...
    ∇mag(d_in, d_res)
}
```
Optimization & Automatic Differentiation

Differentiating after optimization can create asymptotically faster gradients!

\[ O\left(n^2\right) \quad O\left(n\right) \quad O\left(n\right) \]

Optimal for

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

Optimize

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

AD

\[
\text{for } i=n..0 \{ \\
\quad \text{d_res} += \text{d_out}[i]... \\
\}
\]

\[
\nabla\text{mag}(\text{d_in}, \text{d_res})
\]

\[
\text{d_res} = 0.0 \\
\text{for } i=n..0 \{ \\
\quad \text{d_res} += \text{d_out}[i]... \\
\}
\]

\[
\nabla\text{mag}(\text{d_in}, \text{d_res})
\]

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

AD

\[
\text{for } i=n..0 \{ \\
\quad \text{d_res} = \text{d_out}[i]... \\
\quad \nabla\text{mag}(\text{d_in}, \text{d_res}) \\
\}
\]

Optimize

\[
\text{for } i=n..0 \{ \\
\quad \text{d_res} = \text{d_out}[i]... \\
\quad \nabla\text{mag}(\text{d_in}, \text{d_res}) \\
\}
\]
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”
  - Many backends (CPU, CUDA, AMDGPU, etc)
- 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;
}

LLVM

define double @relu3(double %x)

Enzyme Usage

double diffe_relu3(double x) {
    return __enzyme_autodiff(relu3, x);
}
Case Study: ReLU3

Active Instructions

```mlir
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
```
Allocate & zero shadow memory for active values

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

entry

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
```
Compute adjoints for active instructions

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

entry

cond.true

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

cond.end

alloca %result’ = 0.0
alloca %call’ = 0.0
alloca %x’ = 0.0
%cmp = %x > 0
br %cmp, reverse_cond.true, reverse_entry

reverse_cond.true

%df = 3 * pow(%x, 2)
%tmp_call’ = load %call
%x’ += %df * %tmp_call’
store %call’ = 0.0
br reverse_entry

reverse_cond.end

%tmp_res’ = load %result’
%call’ += if %x > 0 then %tmp_res’ else 0
store %result’ = 0.0
br %cmp, reverse_cond.true, reverse_entry

reverse_entry

%0 = load %x’
ret %0
Compute adjoints for active instructions

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

entry

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

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

allocate
%result'
allocate
%call'
allocate
%x'
%cmp = %x > 0
br %cmp, cond.true, cond.end

%df = 3 * pow(%x, 2)
%tmp_call' = load %call
%x' += %df * %tmp_call'
store %call' = 0.0
br reverse_entry

reverse_cond.true

%0 = load %x'
ret %0

reverse_cond.end

reverse_entry
```
Essentially the optimal hand-written gradient!

```c
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];
    dst'[0] = 0;
    src'[1] += dst'[1];
    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]:Pointer, [0,0]:Double, [0,8]:Pointer, [0,8,0]:Integer}
Experimental Setup

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

Enzyme is **4.2x faster** than Reference!
**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

# Create some initial tensor
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)
```

// Input tensor + size, and output tensor
void f(float* inp, size_t n, float* out);

// diffe_dupnoneed specifies not recomputing the output
void diffe(float* inp, float* d_inp, size_t n, float* d_out) {
  __enzyme_autodiff(f, diffe, inp, d_inp, n, diffe_dupnoneed, (float*)0, d_out);
}
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 on CPU
- State-of-the art performance with existing tools
- First general purpose reverse-mode GPU AD
- Novel GPU and AD-specific optimizations improve runtime by several orders of magnitude
- PyTorch-Enzyme & TensorFlow-Enzyme lets researchers use foreign code in ML workflow
Enzyme on the GPU

• Care must be taken to both ensure correctness and maintain parallelism.

• GPU programs have much lower memory limits. Performance is highly dependent on the number of memory transfers.

• Without first running optimizations reverse-mode AD of large kernels is intractable (OOM).

• Novel GPU and AD-specific optimizations can make a difference of several orders of magnitude when computing gradients.

<table>
<thead>
<tr>
<th>Test</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>1</td>
</tr>
<tr>
<td>AD, Optimized</td>
<td>4.4</td>
</tr>
<tr>
<td>AD, No CacheLICM</td>
<td>343.7</td>
</tr>
<tr>
<td>AD, Bad Recompute Heuristic</td>
<td>1275.6</td>
</tr>
<tr>
<td>AD, No Inlining</td>
<td>6372.2</td>
</tr>
<tr>
<td>AD, No PreOptimization</td>
<td>OOM</td>
</tr>
</tbody>
</table>
GPU Gradient Overhead

- Evaluation of both original code and derivative of all inputs (forward or numeric differentiation requires 1 evaluation per input):
  - DG: discontinuous-galerkin (DG) volume integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute bound, respectively)
CUDA Automatic Differentiation

- Enzyme enables differentiation of CPU programs without rewriting them in a DSL.

- Similarly, GPU programs cannot currently be differentiated without being rewritten in a differentiable language (e.g. PyTorch).

- Enzyme enables reverse-mode AD of general existing GPU programs by:
  - Resolving potential data race issues
  - Differentiating parallel control (syncthreads)
  - Differentiating CUDA intrinsics (e.g. threadIdx.x /llvm.nvvm.read.ptx.sreg.tid.x)
  - Handling shared memory
Challenges of Parallel AD

- Benign read race in forward pass => Write race in reverse pass (undefined behavior)

```c
void set(double* ar, double val) {
    parallel_for(int i=0; i<10; i++)
        ar[i] = val;
}
```

```c
double gradient_set(double* ar, double val) {
    double d_val = 0.0;
    parallel_for(int i=0; i<10; i++)
        ar[i] = val;
    parallel_for(int i=0; i<10; i++) {
        d_val += d_ar[i];
        d_ar[i] = 0.0;
    }
    return d_val;
}
```
Parallel Memory Detection

Thread-local memory

• Non-atomic load/store

```
__device__
void f(...) {
    // Thread-local var
double y;
    ...
    d_y += val;
}
```

Same memory location across all threads

• Parallel Reduction

```
// Same var for all threads
double y;
__device__
void f(...) {
    ...
    reduce_add(&d_y, val);
}
```

Others [always legal fallback]

• Atomic increment

```
__device__
// Unknown thread-aliasing
void f(double* y) {
    ...
    atomic { d_y += val; }
}
```
CUDA Automatic Differentiation

%res = load %ptr

store %ptr = %val

%tmp = load %d_res
store %d_res = 0
atomic %d_ptr += %tmp

%tmp = load %d_ptr
store %d_ptr = 0
load/store %d_val += %tmp

- Shadow Registers %d_res and %d_val are **thread-local** as they shadow thread-local registers.
- No risk of races and no special handling required.
- Both %ptr and shadow %d_ptr might be raced upon and require analysis.
Differentiation of SyncThreads

- Sync is only necessary if A and B may write to the same memory
- Four cases for what sync could represent:
  1. All stores in A must complete prior to a load in B
  2. All loads in A must complete prior to a store in B
  3. All stores in A must complete prior to a stores in B [clobber]
  4. All load in A must complete prior to a load in B [unnecessary sync]
CUDA Performance Improvements

- Enzyme may need to cache values from the forward pass for later use in a reverse pass computation
  - When a value needs caching, Enzyme allocates memory (via malloc inside kernel)
  - Potentially quite slow
  - May overwhelm the amount of GPU heap memory

```c
void f(float* in, float* out) {
    float tmp;
    for (int i=0; i<N; i++) {
        tmp = compute(in, i);
        out[i] = tmp * tmp + ...;
    }
}
```

```c
void diffe_f(float* in, float* out) {
    float* tmp_cache = malloc(…);
    for (int i=0; i<N; i++) {
        ...
        tmp_cache[i] = tmp;
    }
    for (int i=N-1; i>=0; i--) {
        ... d_tmp[0] = 2 * tmp_cache[0] * d_out[i];
        d_compute(...);
    }
    free(tmp_cache);
}
```

Value `tmp` is overwritten every iteration and must be cached
Case 1: Store, Sync, Load

codeA(); // store %ptr
sync_threads;
codeB(); // load %ptr
...
diffe_codeB(); // atomicAdd %d_ptr
sync_threads;
diffe_codeA(); // load %d_ptr
    // store %d_ptr = 0

Correct

- Load of d_ptr must happen after all atomicAdds have completed
Case 2: Load, Sync, Store

```c
codeA(); // load %ptr
sync_threads;

codeB(); // store %ptr
...
diffe_codeB(); // load %d_ptr
   // store %d_ptr = 0
sync_threads;
diffe_codeA(); // atomicAdd %d_ptr
```

- All of the stores of d_ptr will complete prior to any atomicAdds

Correct

No cross-thread race here since that’s equivalent to a write race in B
Case 3: Store, Sync, Store

```c
codeA(); // store %ptr
sync_threads;

codeB(); // store %ptr
...

diffe_codeB(); // load %d_ptr
    // store %d_ptr = 0
sync_threads;

diffe_codeA(); // load %d_ptr
    // store %d_ptr = 0
```

Correct

- All stores to d_ptr in diffe_B will complete prior to diffe_A, ensuring only the clobbering store has its derivative incremented
CUDA Automatic Differentiation

- Most CUDA intrinsics [e.g. threadIdx.x] are inactive and recomputable and thus are incorporated into Enzyme without any special handling
- Derivative of syncthreads is a syncthreads at the corresponding place in reverse pass
- Shared memory is handled by making a second shared memory allocation to act as the shadow for any potentially active uses
CUDA Example

```cuda
__device__ void inner(float* a, float* x, float* y) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];
}

__device__ void __enzyme_autodiff( void*, ...);

__global__ void daxpy(float* a, float* da, float* x, float* dx, float* y, float* dy) {
    __enzyme_autodiff((void*)inner, a, da, x, dx, y, dy);
}

__device__ void dffe_inner(float* a, float* da, float* x, float* dx, float* y, float* dy) {
    y[threadIdx.x] = a[0] * x[threadIdx.x];

    float dy = dy[threadIdx.x];
    dy[threadIdx.x] = 0.0f;

    float dx_tmp = a[0] * dy;
    atomic { dx[threadIdx.x] += dx_tmp; }

    float da_tmp = x[threadIdx.x] * dy;
    atomic { da[0] += da_tmp; }
}
```
CUDA Performance Improvements

- Introduce optimizations to reduce the use of memory
  - Alias Analysis to determine legality of recomputing an instruction
    - More aggressive alias analysis properties of syncthreads
  - Don’t cache unnecessary values
    - Move cache outside of loops when possible
  - Heap-to-stack [and to register]
  - Don’t cache memory itself acting as a cache [such as shared memory]
## CUDA Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Forward Pass</th>
<th>Gradient No Opt</th>
<th>+ Standard Opt</th>
<th>+ Cache Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>XSBench-CUDA</strong></td>
<td>1.0s</td>
<td>OOM</td>
<td>20.1s</td>
<td>5.0s</td>
</tr>
<tr>
<td><strong>RSBench-CUDA</strong></td>
<td>1.9s</td>
<td>OOM</td>
<td>&gt;540s</td>
<td>7.8s</td>
</tr>
</tbody>
</table>

Evaluated on a 2080 Super FE
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
- Differentiate GPU kernels
- Open Source (enzyme.mit.edu / github.com/wsmoses/Enzyme)
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow
Acknowledgements

• Thanks to James Bradbury, Alex Chernyakhovsky, Hal Finkel, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsan, 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
- Differentiate GPU kernels
- Open Source (enzyme.mit.edu / github.com/wsmoses/Enzyme)
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow
END
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

```python
// PyTorch rewrite of nbody simulator
import torch

def step(bodies, dt):
    acc = []
    for i in range(len(bodies)):
        acc[i] = torch.zeros([3])
    for i in range(len(bodies)):
        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
```

```cpp
// C++ nbody simulator

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;
    }
}
```
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)
Parallel Memory Detection

- Thread-local memory
  - Non-atomic load/store
  - Same memory location across all threads
  - Parallel Reduction

- Others [always legal fallback]
  - Atomic increment

```c
%tmp = load %d_res
store %d_res = 0
atomic %d_ptr += %tmp
```
Differentiation of SyncThreads

Case 3 [write sync write]

codeA(); // store %ptr
sync_threads;
codeB(); // store %ptr
...
diffe_codeB(); // load %d_ptr
   // store %d_ptr = 0
sync_threads;
diffe_codeA(); // load %d_ptr
   // store %d_ptr = 0

All uses of stores to d_ptr in diffe_B will correctly complete prior to diffe_A

Case 4 [read sync read]

codeA(); // load %ptr
sync_threads;
codeB(); // load %ptr
...
diffe_codeB(); // atomicAdd %d_ptr
sync_threads;
diffe_codeA(); // atomicAdd %d_ptr

Original and differential sync unnecessary and legal to include
CUDA Performance Improvements

- Introduce optimizations to reduce the use of memory
  - Alias Analysis to determine legality of recomputing an instruction
    - More aggressive alias analysis properties of syncthreads
  - Don’t cache unnecessary values
    - Move cache outside of loops when possible
  - Heap-to-stack [and to register]
  - Don’t cache memory itself acting as a cache [such as shared memory]
- PHI Node unwrapping
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