Post-Optimization Automatic Differentiation by Synthesizing LLVM

William S. Moses
wmoses@mit.edu
EuroAD
August 11, 2020

Valentin Churavy
Automatic Differentiation

- Computing the derivatives of functions is necessary component in machine learning (back-propagation, Bayesian inference, uncertainty quantification), scientific computing (modeling, simulation), and other fields
- Writing derivatives of large codebases is intractable
- Existing solutions:
  - Differentiable DSL (TensorFlow, PyTorch, DiffTaichi)
  - Operator-overloading AD (Adept, ADOL-C, JAX)
  - Source-rewriting (Tapenade, ADIC, Zygote)
Operator Overloading vs Source Writing

- **Operator overloading**
  - Provide differentiable versions of existing language constructs
  - May require rewriting to use non-standard language utilities
  - Often dynamic: storing instructions & values of the forward pass in a tape that is later “interpreted” by the reverse pass

- **Source rewriting**
  - Statically analyze program to produce a new gradient function in the source language
  - Requires all differentiated code ahead of time; difficult to use with external libraries
Existing AD Pipelines
Case Study: Vector Normalization

```c
//Compute magnitude in O(n)
double mag(double* x, size_t n);

//Compute norm in O(n^2)
void norm(double* out, double* in, size_t n) {
    for(int i=0; i<n; i++) {
        out[i] = in[i]/mag(in, n);
    }
}
```
double mag(double* x, size_t n);

void norm(double* out, double* in, size_t n) {
    for(int i=0; i<n; i++) {
        out[i] = in[i]/mag(in, n);
    }
}

Loop Invariant Code Motion

double mag(double* x, size_t n);

void norm(double* out, double* in, size_t n) {
    double res = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }
}
void dnorm(double* out, double* dout, 
        double* in, double* din, size_t n) {
    double res = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }
    double d_res = 0;
    for(int i=0; i<n; i++) {
        dres += -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
    }
    dmag(in, din, n, dres);
}
void dnorm(double* out, double* dout,
    double* in, double* din, size_t n) {

    for(int i=0; i<n; i++) {
        out[i] = in[i]/mag(in, n);
    }

    for(int i=0; i<n; i++) {
        double dres = -in[i]*in[i]/mag * dout[i];
        din[i] += dout[i]/mag;
        dmag(in, din, n, dres);
    }
}
void dnorm(double* out, double* dout, double* in, double* din, size_t n) {
    double res = mag(in, n);
    for(int i=0; i<n; i++) {
        out[i] = in[i]/res;
    }
    for(int i=0; i<n; i++) {
        double dres = -in[i]*in[i]/res * dout[i];
        din[i] += dout[i]/res;
        dmag(in, din, n, dres);
    }
}

AD then LICM

\[ O(n) \]

\[ O(n^2) \]

Can’t LICM dmag as it uses loop-local dres
Enzyme Approach

Perform AD on *optimized* programs
Challenges of post-optimization AD

- Implement all optimizations in AD system
- Embed a compiler into your AD
- Rewrite all compiler analyzes and optimizations
- Perform AD on low-level post-optimization representation
- Embed AD into your compiler

“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
Enzyme

- Reverse-mode source-rewriting AD plugin for statically analyzable LLVM IR
- 4.5x speedup over AD before optimization
- State-of-the art performance with existing tools
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- PyTorch-Enzyme / TensorFlow-Enzyme packages to let researchers use foreign code in their ML workflow
- Multisource AD & library support by leveraging LTO
What is LLVM

- Generic low-level compiler infrastructure
- “Cross platform assembly”
- Goal: efficient compilation of arbitrary code
- Well-defined semantics
- Large collection of optimization and analysis passes for handling
LLVM represents each function as a control-flow graph (CFG) of BasicBlocks, containing lists of Instructions.

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n - 1);
    y = fib(n - 2);
    return x + y;
}
```
Core Algorithm

- Type Analysis
- Activity Analysis
- Synthesize derivatives
  - Forward pass that mirrors original code
  - Reverse pass inverts instructions in forward pass (adjoints)
- Optimize
The “memcpy” Problem

- Taking the derivative of operations such as memcpy
  memcpy depends on the type of the data being copied
  - e.g. one derivative for pointers, one for doubles, another for floats
- LLVM Types \(\neq\) C/C++ types
```c
void f(void* dst, void* src) {
    memcpy(dst, src, 8);
}

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;
}

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;
}
```
Type Analysis

- New interprocedural dataflow analysis that detects the underlying type of data
- Each value has a set of memory offsets: type

\[ x = \{[:\text{Pointer}, [0]:\text{Double}, [8]:\text{Pointer}, [8,0]:\text{Integer}}\]
Type Analysis

- Initialize type trees for values from constant, TBAA, and known instruction information
- Each instruction has a type propagation rule describing how types flow through
- Perform series of fixed-point updates propagating type information to uses/users
- Provide a compile-time error if a necessary type cannot be deduced statically
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}

ptr2 = indirect
ptr3 = indirect
```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
TBAA Propagation

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}

ptr2 = indirect
ptr3 = indirect
ptr => cptr2

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}

int* indirect(int* x, int idx) {
    x:  {[]:Pointer, [24]:Int}
    idx: {[]:Int@2}
    &x[idx] {[]:Pointer, [0]:Double}
    return {[]:Pointer, [0]:Double}
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
ptr2 Call IPO - x

```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```
void callee(int* ptr) {
    ptr:      {
        [0]: Pointer, [24]: Int
    }
    ptr2:     {
        [0]: Pointer, [0]: Double
    }
    loadtype: {
        [0]: Double
    }
    ptr3:     {
        [0]: Pointer
    }
    cptr2:    {
        [8]: Int
    }
    notype:   {
        [0]: Pointer
    }
    cptr3:    {
        [0]: Pointer, [0]: Int
    }
}
```

```
ptr2 = indirect
```

```
int* indirect(int* x, int idx) {
    x:      {
        [16]: Pointer, [24]: Double, [24]: Int
    }
    idx:    {
        [0]: Int
    }
    &x[idx]: {
        [0]: Pointer, [0]: Double, [8]: Int
    }
    return  {
        [0]: Pointer, [0]: Double, [8]: Int
    }
```
ptr2 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
void callee(int* ptr) {
    ptr:  {[:,]:Pointer, [16]:Double, [24]:Int}
    ptr2: {[:,]:Pointer, [0]:Double, [8]:Int}
    loadtype: {[:,]:Double}
    ptr3: {}
    cptr2: {[:,]:Pointer, [8]:Int}
    notype: {}
    cptr3: {[:,]:Pointer, [0]:Int}
```
Activity Analysis

- Determines what instructions could impact derivative computation
- Avoids taking meaningless or unnecessary derivatives (e.g. d/dx cpuid)
- Instruction is active iff it can propagate a differential value to its return or memory
- Build off of alias analysis & type analysis
  - 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

- Derivatives of values are stored in shadow allocations
- For all active values, allocate and zero shadow memory to store the derivative of all of its occurrences
- All data structures need to have a shadow data structure created
  - Enzyme will create shadow allocation/stores for structures created inside code being differentiated
  - Data structures passed as arguments will pass shadow arguments
Derivative Synthesis

- Initialize shadow memory
- For each BasicBlock BB:
  - For each Instruction I in reverse(BB):
    - Emit adjoint I, caching and reloading any necessary values from the forward pass
Case Study: ReLU3

define double @relu3(double %x)

double relu3(double x) {
    double result;
    if (x > 0)
        result = pow(x, 3);
    else
        result = 0;
    return result;
}

double diffe_relu3(double x) {
    return __enzyme_autodiff(relu3, x);
}
Case Study: ReLU-f

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

Allocate & zero shadow memory for active instructions

Allocate:
- `%result' = 0.0`
- `%call' = 0.0`
- `%x' = 0.0`

Condition (`cond.true`):
- `%cmp = %x > 0`
- `br %cmp, cond.true, cond.end`

Computation:
- `%call = pow(%x, 3)`
- `br cond.end`

Result:
- `%result = phi [%call, cond.true], [0, entry]`
- `; deleted return`
- `%result' = 1.0`
- `br reverse_cond.end`
define double @diffe_relu3(double %x, double %differet)

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

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
Essentially the optimal hand-compiled program!

define double @diffe_relu3(double %x)

double diffe_relu3(double x) {
  double result;
  if (x > 0)
    result = 3 * pow(x, 2);
  else
    result = 0;
  return result;
}
Adjoint instructions may require values from the forward pass

- e.g. \( \nabla(x \times y) \Rightarrow x \frac{dy}{dx} + y \frac{dx}{dx} \)

For all such values, allocate memory in the function header to store the value for use in the reverse pass.

Values computed inside loops are stored in an array indexed by the loop induction variable.

- Array allocated statically if possible; otherwise dynamically `realloc`d
Case Study: Read Sum

double sum(double* x) {
    double total = 0;
    for(int i=0; i<10; i++)
        total += read() * x[i];
    return total;
}

#define double @sum(double* %x)

define double @sum(double* %x)

double sum(double* x) {
    double total = 0;
    for(int i=0; i<10; i++)
        total += read() * x[i];
    return total;
}

void diffe_sum(double* x, double* xp) {
    return __enzyme_autodiff(sum, x, xp);
}

void diffe_sum(double* x, double* xp) {
    return __enzyme_autodiff(sum, x, xp);
}
Case Study: Read Sum

Active Variables

```c
define double @sum(double* %x)

%result = phi [%call, cond.true], [0, entry]
ret %result
```

```c
%i = phi [0, entry], [%i.next, for.body]
%total = phi [0.0, %entry], [%add, for.body]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body
```
Case Study: Read Sum

Each register in the for loop represents a distinct active variable every iteration.

```plaintext
define double @sum(double* %x)

entry
br for.body

for.body
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body

for.cleanup
%result = phi [%call, cond.true], [0, entry]
ret %result
```
Define double @diffe_sum(double* %x, double* %xp)

Allocate & zero shadow memory per active value

Allocate %x' = 0.0
Allocate %total' = 0.0
Allocate %0' = 0.0
Allocate %mul' = 0.0
Allocate %add' = 0.0
Allocate %result' = 0.0

Branch for.body

%i = phi [0, entry], [%i.next, for.body]
%total = phi [0.0, %entry], [%add, for.body]
%call = @read()
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
Branch %exitcond, for.cleanup, for.body

%result = phi [%call, cond.true], [0, entry]
Return %result
Cache forward pass variables for use in reverse

```c
define double @diffe_sum(double* %x, double* %xp)
entry
alloca %x' = 0.0
alloca %total' = 0.0
alloca %0' = 0.0
alloca %mul' = 0.0
alloca %add' = 0.0
alloca %result' = 0.0
%call_cache = @malloc(10 x double)
br for.body

for.body
%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
store %call_cache[%i] = %call
%0 = load %x[%i]
%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body

for.cleanup
%result = phi [ %call, cond.true], [0, entry]
@free(%cache)
ret %result
```
define void @diffe_sum(double* %x, double* %xp)

entry

%call_cache = @malloc(10 x double)
br for.body

for.body

%i = phi [ 0, entry ], [ %i.next, for.body ]
%total = phi [ 0.0, %entry ], [ %add, for.body ]
%call = @read()
store %call_cache[%i] = %call
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, reversefor.body, for.body

reversefor.body

%i' = phi [ 9, for.body ], [ %i’.next, reversefor.body ]
%i’.next = %i’ - 1
%cached_read = load %call_cache[%i’]
store %xp[%i’] = %cached_read + %xp[%i’]
%exit2 = %i = 0
br %exitcond, %exit2, reversefor.body

exit

@free(%cache)
ret

After lowering & some optimizations
define void @diffe_sum(double* %x, double* %xp)

%call0 = @read()
store %xp[0] = %call0
%call1 = @read()
store %xp[1] = %call1
%call2 = @read()
store %xp[2] = %call2
%call3 = @read()
store %xp[3] = %call3
%call4 = @read()
store %xp[4] = %call4
%call5 = @read()
store %xp[5] = %call5
%call6 = @read()
store %xp[6] = %call6
%call7 = @read()
store %xp[7] = %call7
%call8 = @read()
store %xp[8] = %call8
%call9 = @read()
store %xp[9] = %call9
ret

After more optimizations

void diffé_sum(double* x, double* xp) {
    xp[0] = read();
    xp[1] = read();
    xp[2] = read();
    xp[3] = read();
    xp[4] = read();
    xp[5] = read();
    xp[6] = read();
    xp[7] = read();
    xp[8] = read();
    xp[9] = read();
}
Cache Optimizations

❖ By carefully caching in a form LLVM understands, existing optimization passes can optimize the memory away! [*]

❖ Further optimizations:
  ❖ Use alias analysis to prove that recomputing an instruction is legal
  ❖ Don’t cache unnecessary values
  ❖ Don’t cache a value that already has already been cached elsewhere

[*] For dynamic loops, requires modification to LLVM memory analyses to understand semantics of realloc.
Function Calls

- Computing both forward and reverse pass in the same function allows further optimization and reduces memory usage.
  - Enzyme uses Alias Analysis to detect legality of computing forward/reverse pass together.
  - Otherwise, Enzyme may need to modify forward pass to cache values needed by reverse pass.
Indirect Function Calls

- Calls to functions that aren’t known at compile time are dealt with by leveraging shadow memory.
- The shadow of function pointers is defined to be a global containing the forward and reverse pass.
- Thus taking the adjoint of an indirect function call simply requires extracting and calling the corresponding shadow callee.
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
Evaluation

- Collection of benchmarks from Microsoft’s ADBench suite and of technically interest
- Evaluated Enzyme, Reference, and the two fastest AD systems from ADBench (Tapenade, Adept)
- All programs run serially
- Quiesed Amazon c4.8xlarge (disabled turbo-boost; hyper-threading)
Relative Speedup

Higher is Better

Speedup of 0.5 denotes program took twice as long as Speedup of 1.0
<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Ref</th>
<th>Tapenade</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>2.353</td>
<td>4.458</td>
<td>4.042</td>
<td>7.645</td>
</tr>
<tr>
<td>BA</td>
<td>0.424</td>
<td>0.778</td>
<td>0.680</td>
<td>2.334</td>
</tr>
<tr>
<td>GMM</td>
<td>0.073</td>
<td>0.462</td>
<td>0.124</td>
<td>1.544</td>
</tr>
<tr>
<td>Euler</td>
<td>0.161</td>
<td>36.723</td>
<td>nan</td>
<td>6.851</td>
</tr>
<tr>
<td>RK4</td>
<td>3.397</td>
<td>23.442</td>
<td>nan</td>
<td>6.371</td>
</tr>
<tr>
<td>FFT</td>
<td>0.183</td>
<td>0.182</td>
<td>nan</td>
<td>2.538</td>
</tr>
<tr>
<td>Bruss</td>
<td>0.181</td>
<td>0.182</td>
<td>0.518</td>
<td>3.457</td>
</tr>
</tbody>
</table>

Enzyme is 4.5x faster than Ref!
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)

import tensorflow as tf
from tf_enzyme import enzyme

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_dunpnoneed specifies not recomputing the output
void diffe(float* inp, float* d_inp, size_t n, float* d_out) {
    __enzyme_autodiff(f, diffe_dup, inp, d_inp, n, diffe_dunpnoneed, (float*)0, d_out);
}
Conclusions

- AD on low-level IR can be performant
- Optimization before AD is crucial
- Enzyme provides high-performance cross-language AD
- Open-sourcing late summer (email for beta access!)

Future Work:

- Parallelism, GPU AD
- AD-specific optimizations
Acknowledgements

❖ Thanks to James Bradbury, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Yo Shavit, Dhash Shrivathsa, Nalini Singh, and Alex Zinenko for their invaluable feedback and advice.

❖ William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323.

❖ 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.
Conclusions

- AD on low-level IR can be performant
- Optimization before AD is crucial
- Enzyme provides high-performance cross-language AD
- Open-sourcing late summer (email for beta access!)
- Future Work:
  - Parallelism, GPU AD
  - AD-specific optimizations
Backup Slides
Requirements & Performance Boosts

- **Requirements**
  - Enable TBAA (Type based alias analysis)
  - Strict Aliasing (no unions)
  - Disable exceptions

- **Performance Boosts**
  - Disable Loop Unrolling before AD
  - Disable Vectorization before AD
Future Work: Parallelism

- Build off prior work [1] representing parallelism (OpenMP, Cilk, etc) in compiler
- Reverse pass can remain in parallel, with dependencies reversed
- Updates to adjoints in parallel tasks done with reducer or atomic add to prevent races


[*] Work in progress — suggestions appreciated
Benchmarks

- LSTM: Long-short term memory model
- BA: Bundle analysis
- GMM: Gaussian mixture model
- Euler: Euler integration
- RK4: Runge-Kutta integration
- FFT: Fast Fourier transform
- Bruss: Brusselrator chemical simulation
# Matrix Vector: Single Iteration

```c
#define N 20000
#define M 20000
#define ITERS 1
```

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>1.119</td>
<td>0.0006</td>
</tr>
<tr>
<td>Forward</td>
<td>1.119</td>
<td>11.016</td>
</tr>
<tr>
<td>Forward + Reverse</td>
<td>1.210</td>
<td>13.445</td>
</tr>
</tbody>
</table>
Taylor Expand Log

static adouble logger(adouble x) {
    adouble sum = 0;
    for(int i=1; i<=ITERS; i++) {
        sum += pow(x, i) / i;
    }
    return sum;
}

static double logger_and_gradient(double xin, double& xgrad) {
    adept::Stack stack;
    adouble x = xin;
    stack.new_recording();
    adouble y = logger(x);
    y.set_gradient(1.0);
    stack.compute_adjoint();
    xgrad = x.get_gradient();
    return y.value();
}
Taylor Expand Log (Julia)

\[ f(x) = \sum_{i=1}^{N} \frac{x^i}{i} \approx -\log(1 - x) \]

```plaintext
#define ITERS 10000000
double logger(double x) {
    double sum = 0;
    for(int i=1; i<=ITERS; i++)
        sum += pow(x, i) / i;
    return sum;
}
```

```plaintext
function jl_f1(f::Float64)
    sum = 0 * f;
    for i = 1:10000000
        sum += f^i / i;
    end
    return sum;
end
```

\[ \frac{\partial}{\partial x} f(x) \approx \frac{1}{1 - x} \]

\[ \frac{\partial}{\partial x} f(x = 0.5) \approx 2 \]

```plaintext
using Zygote
@show autodiff(jl_f1, 0.5)
@time autodiff(jl_f1, 0.5)
```

```plaintext
; Enzyme derivative code
@show autodiff(fl_f1, 0.5)
@time autodiff(fl_f1, 0.5)
```

```plaintext
using Zygote
@show jlf1′(0.5)
@time jlf1′(0.5)
```
## Taylor Expand Log

10000000 iterations

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
<th>Enzyme-Julia</th>
<th>Zygote-Julia</th>
<th>AutoGrad-Julia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.74</td>
<td>3.72</td>
<td>3.82</td>
<td>3.82</td>
<td>3.82</td>
</tr>
<tr>
<td>Forward</td>
<td>3.74</td>
<td>4.56</td>
<td>3.82</td>
<td>3.82</td>
<td>3.82</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>3.90</td>
<td>4.65</td>
<td>3.95</td>
<td>44.694</td>
<td>896.30</td>
</tr>
</tbody>
</table>
#define N 10000000

double logsumexp(double* x, size_t n) {
    double A = 0;
    for(int i=1; i < n; i++) {
        A = max(A, x[i]);
    }
    double sema = 0;
    for(int i=0; i < n; i++) {
        sema += max(x[i] - A);
    }
    return max(sema) + A;
}

function logsumexp(x::Array{Float64,1})
    A = maximum(x)
    ema = exp.(x .- A)
    sema = sum(ema)
    return log(sema) + A
end
# Taylor Expand Log

10000000 iterations

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.74</td>
<td>3.72</td>
</tr>
<tr>
<td>Forward</td>
<td>3.74</td>
<td>4.56</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>3.90</td>
<td>4.65</td>
</tr>
</tbody>
</table>
## LogSumExp

100,000,000 elements

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.364</td>
<td>0.364</td>
</tr>
<tr>
<td>Forward</td>
<td>0.364</td>
<td>2.994</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>0.605</td>
<td>3.836</td>
</tr>
</tbody>
</table>
Find Matrix by Gradient Descent

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>4.731</td>
<td>25.606</td>
</tr>
<tr>
<td>Gradient Descent</td>
<td>22.672</td>
<td>133.354</td>
</tr>
</tbody>
</table>
Training Simple Neural Network

<table>
<thead>
<tr>
<th>Enzyme</th>
<th>Adept</th>
<th>Handwritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>73.718</td>
<td>338.097</td>
<td>72.178</td>
</tr>
</tbody>
</table>

Picked first C MNIST Code on Github:
https://github.com/AndrewCarterUK/mnist-neural-network-plain-c

❖ 1-layer fully connected layer => softmax => cross-entropy loss
❖ Batch size 100
❖ 1000 iterations
❖ Learning rate 0.5
Case Study: Subcall

double loadsq(double* x) {
    return x[0] * x[0];
}

void f(double* x) {
    *x = loadsq(x);
}

void diffe_f(double* x, double* xp) {
    __enzyme_autodiff(f, x, xp);
}

define double @loadsq(double* %x)

define void @f(double* %x)

define void @diffe_f(double* %x, double* %xp)
double loadsq(double* x) {
    return x[0] * x[0];
}

void f(double* x) {
    *x = loadsq(x);
}

define {double,double} @augment_loadsq(double* %x)

entry
%val = load %x
%mul = %val * %val
ret {/*return val*/%mul,
     /*cache*/%val}

define void @diffe_loadsq(double* %x, double* %x’, double %diffe, double %cache)

entry
%val = %cache // cannot reload as x changed
%mul = %val * %val
%mul' = %diffe
%val' = 2 * %val * %mul'
store %x’ += %val’
define {double, double} @augment_loadsq(double* %x)

entry
%val = load %x
%mul = %val * %val
ret {/*return val*/%mul,
    /*cache*/%val}

double loadsq(double* x) {
    return x[0] * x[0];
}

void f(double* x) {
    *x = loadsq(x);
}

define void @diffe_loadsq(double* %x, double* %x’, double %diffe, double %cache)

entry
%val = %cache // cannot reload as x changed
%mul = %val * %val
%mul’ = %diffe
%val’ = 2 * %val * %mul’
store %x’ += %val’

define void @diffe_f(double* %x)

entry
{%call, %cache} = @augment_loadsq(%x)

store %x = %call
%call’ = load %x’
store %x’ = 0
@augment_loadsq(%x, %x’, %call’, %cache)
ret
double loadsq(double* x) {
    return x[0] * x[0];
}

void f(double* x) {
    *x = loadsq(x);
}

#define {double, double} @augment_loadsq(double* %x)

entry %val = load %x
%mul = %val * %val
ret { /*return val*/ %mul,
     /*cache*/ %val}

#define void @diffe_loadsq(double* %x', double %diffe, double %cache)

entry store %x' += 2 * %cache * %diffe

#define void @diffe_f(double* %x)

entry { %call, %cache } = @augment_loadsq(%x)
store %x = %call

%call' = load %x'
store %x' = 0

@augment_loadsq(%x', %call', %cache)
ret
ptr2 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
void callee(int* ptr) {
    ptr:  {[]}::Pointer, [16]:Double, [24]:Int
    ptr2:  {[]}::Pointer, [0]:Double, [8]:Int
    loadtype: {[]}::Double
    ptr3:  {}
    cptr2:  {[]}::Pointer, [8]:Int
    notype:  {}
    cptr3:  {[]}::Pointer, [0]:Int
```

```c
int* indirect(int* x, int idx) {
    x:  {[]}::Pointer, [16]:Double, [24]:Int
    idx:  {[]}::Int@2
    &x[idx] {[]}::Pointer, [0]:Double, [8]:Int
    return  {[]}::Pointer, [0]:Double, [8]:Int
```
ptr => cptr2

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```
void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
ptr3 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```c
void callee(int* ptr) {
    ptr: {
        [1]: Pointer, [16]: Double, [24]: Int
    }
    ptr2: {
        [0]: Pointer, [8]: Int
    }
    loadtype: {
        [0]: Double
    }
    ptr3: {}
    cptr2: {
        [0]: Pointer, [8]: Int
    }
    notype: {
        [8]: Int
    }
    cptr3: {
        [0]: Pointer, [16]: Double
    }
}
```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
ptr3 Call IPO - return

callee:

```c
void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}
```

ptr3 = indirect

```c
int* indirect(int* x, int idx) {
    x:   {[]}::Pointer, [16]:Double, [24]:Int
    idx: {[]}::Int@3
    &x[idx] {[]}::Pointer, [0]:Int
    return {[]}::Pointer, [0]:Int
}
```
ptr3 Call IPO

```c
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}
```

callee:

```c
void callee(int* ptr) {
    ptr: {[]:Pointer, [16]:Double, [24]:Int}
    ptr2: {[]:Pointer, [0]:Double, [8]:Int}
    loadtype: {[]:Double}
    ptr3: {[]:Pointer, [0]:Int}
    cptr2: {[]:Pointer, [0]:Double, [8]:Int}
    notype: {[]:Double}
    cptr3: {[]:Pointer, [0]:Int}
}
```
int* indirect(int* x, int idx) {
    return &x[idx];
}

void callee(int* ptr) {
    int* ptr2 = indirect(ptr, 2);
    double loadtype = *(double*)ptr2;
    int* ptr3 = indirect(ptr, 3);
    int* cptr2 = &ptr[2];
    int notype = *cptr2;
    int* cptr3 = &ptr[3];
    *((int64_t*)cptr3) = 100;
}