Enzyme: Efficient Cross-Platform AD by Synthesizing LLVM

William S. Moses
Tim Kaler

{wmoses, tfk}@mit.edu
EuroAD
July 2, 2019
Part of exploration on AD by MIT Supertech Research Groups

https://supertech.mit.edu

Charles E. Leiserson  Tao B. Schardl  Daniele Vettorel

Funding provided by DOE CSGF fellowship and IBM
“Holy Grail” of Automatic Differentiation

- **General**: we should be able to run AD on arbitrary programs
- **Easy to Use**: the amount of code one needs to modify to use AD should be small
- **Fast**: executing AD shouldn’t take too long
- **Correct**: AD should produce the right answer
State of AD

<table>
<thead>
<tr>
<th>Level</th>
<th>Usable</th>
<th>Fast</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library/DSL Specific</td>
<td>✔️</td>
<td>✔️</td>
<td>Zygote</td>
</tr>
<tr>
<td>High Level Lang. Specific</td>
<td>✔️</td>
<td>❓</td>
<td>Adept</td>
</tr>
<tr>
<td>Low Level Lang. Specific</td>
<td>❌</td>
<td>❓</td>
<td>None*</td>
</tr>
<tr>
<td>Language Independent</td>
<td>❌</td>
<td>❓</td>
<td></td>
</tr>
</tbody>
</table>

*There are some “language independent” ones but they require rewriting for said framework in a way that makes it rather unusable.
State of AD

- Library/DSL Specific
  - Usable: Yes
  - Fast: Yes
  - Examples: Enzyme

- High Level Lang. Specific
  - Usable: Yes
  - Fast: Unknown
  - Examples: Zygote

- Low Level Lang. Specific
  - Usable: No
  - Fast: Unknown
  - Examples: Adept

*There are some “language independent” ones but they require rewriting for said framework in a way that makes it rather unusable.
Why Generality Matters

- Taking derivatives of arbitrary programs gives programmers composability — they only need to care about the tool they’re building rather than any code they’re differentiating
  - e.g. ‘I want to build ML tool for predicting the result of this simulator’
- Most programs aren’t written in the same language / framework as your tool and thus won’t work with your AD
Idea: Generality by Bootstrapping

- A sufficiently general AD system for a particular language (or framework) works not only with code in that language, but any code for higher level languages written in the lower level language.

- i.e a good C differentiator should be able to also differentiate Python code

- If we create a general AD for a low level language we get the higher languages (mostly) for free
Presenting Enzyme (work in progress)

- Reverse-mode automatic differentiation tool built in LLVM to handle a variety of languages and frameworks
- Performs differentiation by synthesizing a new function
- Clean interface that doesn’t require rewriting existing programs to use
Presenting Enzyme (work in progress)

- Reverse-mode automatic differentiation tool built in LLVM to handle a variety of languages and frameworks
- Performs differentiation by synthesizing a new function
- Clean interface that doesn’t require rewriting existing programs to use

Our beta* can match the performance of less general AD on a variety of benchmarks!

[*] Beta is in progress and not yet feature-complete. Planned open sourcing once published and ABI-stable.
Lowering pass needs to be implemented for each language.
C/C++ and Julia implemented presently.

[1] Frontend for Julia joint with Valentin Churavy
[2] Lowering pass needs to be implemented for each language.
C/C++ and Julia implemented presently.
What is LLVM

- Generic low-level compiler infrastructure
  - “Cross platform assembly”
- Goal is compiling arbitrary code as efficiently as possible
  - Well-defined semantics and high-level constructs
  - 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;
}
```
What is Synthesis?

- Instrumentation-based approaches
  - Store the operations and values of the forward pass in a tape that is later “interpreted” by the reverse pass
  - Can store data by overloading a language’s types/functions or rewriting/transforming the source code to include it (such as in compiler instrumentation)

- Synthesis-based approaches
  - Statically analyze the function to produce a new function with the relevant operations
Why Synthesis?

- Synthesis is often faster
- Overloading all of LLVM’s instructions and fixing its ~4 million lines of code is both impractical and unsustainable
- Since we must do program rewriting/ transformation anyways (and LLVM has tools for making it easier), might as well do synthesis rather than instrumentation for faster results
Core Algorithm

- Iterate through all instructions in the original function to detect whether they are active (could modify derivatives) or not.

- For active value in the original function, allocate and zero memory to store the derivative of all of its occurrences.

- For each block in the original function, compute the adjoint of its active instructions in reverse order, caching and reloading any necessary values from the forward pass.
Optimizing away the “Tape”

- To compute adjoints, it may be necessary to use values computed in the forward pass
  - Traditionally stored in a stack-based tape mechanism
- Idea: carefully cache all values individually in a form LLVM understands (to simplify implementation)
  - Existing optimization passes can optimize the memory away
- Without optimization may use more memory than traditional tape, after optimization uses far less
Optimizing Communication

- Compute the forward pass and backward passes together
  - Let LLVM optimize how values are shared/reused from forward to backward pass
  - Dead code-elimination can get rid of the forward pass if not needed!
- After optimizations, forward pass and backward pass can be split* [useful for recursive calls]

[*] Splitting is in progress.
Case Study: ReLU-f

```c
double reluf(double x) {
    double result;
    if (x > 0)
        result = f(x);
    else
        result = 0;
    return result;
}
```

```c
double diffe_reluf(double x) {
    return __builtin_autodiff(reluf, x);
}
```
define double @diffe_reluf(double %x)

alloca %result' = 1.0
alloca %call' = 0.0
alloca %x' = 0.0
%cmp = %x < 0
br %cmp, cond.true, cond.end

%call = f(%x)
br cond.end

%result = phi [ %call, cond.true], [0, entry]
br reverse_cond.end

%8 = %x < 0
%9 = load %call'
%10 = load %result'
%11 = if %8 then %10 else %9
store %call' = %11
store %result' = 0.0
br %cmp, reverse_cond.true, reverse_entry

%0 = load %x'
ret %0

reverse_entry

reverse_cond.true

%3 = diffe_f(%x)
%4 = load %call'
%5 = %4 * %3
%6 = load %x'
%7 = %6 + %5
store %x' = %7
store %call' = 0.0
br reverse_entry

cond.true

entry

reverse_cond.end

cond.end

reverse_entry
define double @diffe_reluf(double %x)

%cmp = %x < 0
br %cmp, cond.true, cond.end

call = f(%x)
br cond.end

%result = phi [ %call, cond.true], [0, entry]
br reverse_cond.end

%8 = %x < 0
%11 = if %8 then 1.0 else 0.0
br %cmp, reverse_cond.true, reverse_entry

%3 = diffe_f(%x)
%5 = %11 * %3
%7 = 0.0 + %5
br reverse_entry

%0 = phi [ %7, reverse_cond.true], [0, reverse_cond.end]
ret %0
define double @diffe_reluf(double %x)

entry

%cmp = %x < 0
br %cmp, cond.true, cond.end

cond.true

%call = f(%x)
br cond.end

%result = phi [ %call, cond.true], [0, entry]
br reverse_cond.end

cond.end

reverse_cond.true

%8 = %x < 0
%11 = if %cmp then 1.0 else 0.0
br %cmp, reverse_cond.true, reverse_entry

%3 = diffe_f(%x)
%5 = 1.0 * %3
%7 = 0.0 + %3
br reverse_entry

reverse_cond.end

%0 = phi [ %3, reverse_cond.true], [0, reverse_cond.end]
ret %0

reverse_entry

Common Sub-expression Elimination & Instruction Simplification
define double @diffe_reluf(double %x)

%cmp = %x < 0
br %cmp, cond.true, cond.end

br cond.end

cond.true

br cond.end

br reverse_cond.end

reverse_cond.true

%3 = diffe_f(%x)
br reverse_entry

br %cmp, reverse_cond.true, reverse_entry

reverse_cond.end

reverse_cond.true

%0 = phi [%3, reverse_cond.true], [0, reverse_cond.end]
ret %0

reverse_entry

cond.end

Dead Code Elimination
Essentially the optimal hand-compiled program!
More Advanced Details
Loops require special handling since an SSA Value can have multiple distinct realizations per iteration of the loop.

Idea: Statically allocate an array of sufficient size to store all loop allocations in outermost loop preheater.

With correct attributes, LLVM is able to understand this allocation and similarly optimize.

If loop bounds cannot be calculated statically, dynamically reallocate array.

Requires modification to LLVM memory analyses to understand semantics of realloc.
Active Variable Detection*

- All function arguments are denoted as either inactive, active (with reasonable defaults for the user)
- Non-pointer value is inactive if it is created by using only inactive values or never used in creation of an active value
- Pointer values require examining stores to uses/users
- Algorithm as heuristic to avoid creating unnecessary computation / synthesis and avoid asking for ill-defined derivatives (i.e. a function prints an active variable — what is the derivative of the print function)

[*] Work in progress — suggestions appreciated
Complex Data Types

❖ Calling a derivative function with complex data types (e.g. arrays) requires passing a second data structure to store derivative outputs

❖ Structs with multiple elements may contain both active variables and constants

❖ e.g. an array storing its size — size is constant

❖ Variable marked as active

❖ Rely on active variable detection to identify if a particular element of struct derivatives
Local Data Structures

- Local data structures with active variable need to be duplicated to store derivative information.
- Leverage all data structures are created by specific memory instructions (malloc/free/new/delete/etc).
- Allocations are copied in forward pass to create differential structures.
- Frees are delayed until the reversed version of the block that allocated in case values are used in the reverse pass.
Case Study: Read Sum

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 __builtin_autodiff(sum, x, xp);
}

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

entry
%cache = @malloc(8 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 %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 %cache[%i']
store %xp[%i'] = %cached_read + %xp[%i']
%exit2 = %i = 0
br %exitcond, %exit2, reversefor.body

exit @free(%cache)
ret

After lowering and some optimizations
Case Study: Read Sum

```c
void @diffe_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();
}
```
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

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

[*] Work in progress — suggestions appreciated
Custom Derivatives*

- Enzyme can compute derivatives of any function in current compilation module
- Functions compiled in a unit (i.e. libraries, linked objects) can be handled by compiling library with Enzyme, creating library with derivatives included
- Functions can be marked with a custom derivative function via metadata

[*] Work in progress — suggestions appreciated
Preliminary Tests of Beta Implementation

All programs run serially

Intel E5520 @ 2.27GHz, 64GB 1066MHz DDR3, Ubuntu 16.04
Time Graph

Between 19% and 534% speedup over Adept
Comparable with Tapenade

Time (normalized)

- TaylorLog
- LogSumExp
- Matrix
- NeuralNet

- Enzyme
- Tapenade
- Adept
Use a Taylor series to compute the log function, evaluated at $x=0.5$

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

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

double derivative(double x) {
    return __builtin_autodiff(taylor_log, x);
}
```
## Taylor Expand Log

100,000,000 iterations

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
<th>Tapenade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>3.71</td>
<td>3.74</td>
<td>3.70</td>
</tr>
<tr>
<td>Forward</td>
<td>3.70</td>
<td>4.50</td>
<td>3.71</td>
</tr>
<tr>
<td>Forward + Reverse</td>
<td>3.72</td>
<td>4.67</td>
<td>3.70</td>
</tr>
</tbody>
</table>
LogSumExp

Smooth approximation to maximum function, often used in machine learning.

```c
#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;
}
```

double derivative(double* input, double* inputp, size_t n) {
    return __builtin_autodiff(logsumexp, input, inputp, n);
}
LogSumExp

10000000 elements

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
<th>Tapenade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.364</td>
<td>0.364</td>
<td>0.364</td>
</tr>
<tr>
<td>Forward</td>
<td>0.364</td>
<td>2.994</td>
<td>0.364</td>
</tr>
<tr>
<td>Forward +Reverse</td>
<td>0.605</td>
<td>3.836</td>
<td>0.817</td>
</tr>
</tbody>
</table>
Find Matrix by Gradient Descent

Find a matrix that produces a vector close to zero when multiplied by vec

```c
#define N 2000
#define M 2000

double matvec(double* mat, double* vec) {
    double* out = malloc(sizeof(double)*N);

    double A = 0;
    for(int i=1; i < N; i++) {
        out[i] = 0;
        for(int j=1; j < M; j++) {
            out[i] += mat[i*M+j] * vec[j];
        }
    }
    double sum = 0;
    for(int i=0; i < N; i++) {
        sum += out[i] * out[i];
    }
    free(out);
    return sum;
}
```

```c
#define ITERS 1000
#define RATE 0.00000001

double descent(double* mat, double* dmat, double* vec) {

    for(int iter=1; iter < ITERS; iter++) {
        memset(dmat, 0, sizeof(double)*N*M);
        __builtin_autodiff(matvec, mat, dmat, differ_const, vec);

        out[i] = 0;
        for(int i=1; i < N*M; i++) {
            mat[i] -= dmat[i] * RATE;
        }
    }
    double sum = 0;
    for(int i=0; i < N*M; i++) {
        mat[i] -= dmat[i] * RATE;
    }
    double sum = 0;
    for(int i=0; i < N; i++) {
        sum += out[i] * out[i];
    }
}
```
Find Matrix by Gradient Descent

<table>
<thead>
<tr>
<th></th>
<th>Enzyme</th>
<th>Adept</th>
<th>Tapenade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>4.698</td>
<td>25.356</td>
<td>4.704</td>
</tr>
<tr>
<td>Gradient Descent</td>
<td>22.039</td>
<td>130.957</td>
<td>21.828</td>
</tr>
</tbody>
</table>
Training Simple Neural Network

<table>
<thead>
<tr>
<th>Enzyme</th>
<th>Adept</th>
<th>Tapenade</th>
<th>Handwritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>73.663</td>
<td>338.097</td>
<td>73.008</td>
<td>72.076</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
Conclusions

❖ Need four things in AD: generality, usability, speed, and correctness

❖ Created a prototype tool: Enzyme
  ❖ Provides first “true” cross platform AD (to our knowledge)
    ❖ Compatible with any tool lowering to LLVM (Tensorflow, Rust, C/C++, Julia, etc)
  ❖ Matches state of art performance by building off on compiler optimizations
  ❖ Demonstrates possibility of a general AD that is efficient and easy-to-use

❖ Future Work:
  ❖ Feature completion and more frontends
  ❖ Heuristics (e.g. recompute vs cache)
  ❖ ABI stability and open source release / publication
Backup Slides
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>
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)

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

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

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

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

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

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

```julia
using Zygote
@show jl_f1′(0.5)
@time jl_f1′(0.5)
```
# Taylor Expand Log

## 10000000 iterations

<table>
<thead>
<tr>
<th>Normal</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>
LogSumExp

```c
#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;
}
```

```python
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

10,000,000 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