Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients!

William S. Moses  Valentin Churavy

wmoses@mit.edu
June 16, 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

![Diagram showing existing automatic differentiation pipelines with languages like C++, Julia, R, and Swift and tools like LLVM andCodeGen.]
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

//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

$O(n^2)$

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

**Optimize**

$O(n)$

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

**AD**

```
d_res = 0.0
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)
}

Optimize

\[ O(n) \]

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

AD

\[ O(n) \]

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

\[ \nabla \text{mag}(d_{in}, d_{res}) \]

\[ O(n^2) \]

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

AD

\[ O(n^2) \]

for i=n..0 {
    d_res = d_out[i]...
    \nabla \text{mag}(d_{in}, d_{res})
}
Optimization & Automatic Differentiation

$O(n^2)$

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

for $i=0..n$ {
    out[$i$] /= res
}

$d_{res} = 0.0$
for $i=n..0$ {
    d_res += d_out[$i$]...
}
\[\nabla \text{mag}(d_{in}, d_{res})\]

$O(n)$

res = mag(in)

$d_{res} = 0.0$
for $i=n..0$ {
    d_res += d_out[$i$]...
}
\[\nabla \text{mag}(d_{in}, d_{res})\]

$O(n^2)$

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

for $i=n..0$ {
    d_res = d_out[$i$]...
}
\[\nabla \text{mag}(d_{in}, d_{res})\]
Optimization & Automatic Differentiation

Differentiating after optimization can create \textit{asymptotically faster} gradients!

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

\begin{verbatim}
for i=0..n {
    out[i] /= mag(in)
}
\end{verbatim}

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

\[ O\left(n\right) \]

\begin{verbatim}
res = mag(in)
for i=0..n {
    out[i] /= res
}
\end{verbatim}

\[ O\left(n\right) \]

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

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

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

\begin{verbatim}
for i=n..0 {
    d_res = d_out[i]...
    ∇mag(d_in, d_res)
}
\end{verbatim}

\[ O\left(n^2\right) \]
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;
}

Enzyme Usage

double diffe_relu3(double x) {
    return __enzyme_autodiff(relu3, x);
}

LLVM

define double @relu3(double %x)

entry

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

cond.true

%call = pow(%x, 3)
br cond.end

cond.end

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

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

entry

cond.true

%call = pow(%x, 3)
br cond.end

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

alloca %result' = 0.0
alloca %call' = 0.0
alloca %x' = 0.0

%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

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

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

alloca %result’ = 0.0
alloca %call’ = 0.0
alloca %x’ = 0.0
%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

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

%0 = load %x’
ret %0

reverse_entry

reverse_cond.true

reverse_cond.end

Compute adjoints for active instructions

18
define double @diffe_relu3(double %x, double %differet)

Compute adjoints for active instructions

entry

cond.true

%call = pow(%x, 3)
br cond.end

alloca %result' = 0.0
alloca %call' = 0.0
alloca %x' = 0.0
%cmp = %x > 0
br %cmp, cond.true, 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

%call = pow(%x, 3)
br 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
Essentially the optimal hand-written gradient!

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;
}
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);
}

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
- Perform series of fixed-point updates through instructions

```
struct MyType {
    double;
    int*;
}
x = MyType*;
```

types(x) = {[0]:Pointer, [0,0]:Double, [0,8]:Pointer, [0,8,0]:Integer}
Cache

- Adjoint instructions may require values from the forward pass
  - e.g. $\nabla(x * y) \Rightarrow x \frac{dy}{dx} + y \frac{dx}{dx}$

- For all values needed in the reverse, allocate memory in the forward pass to store the value

- Values computed inside loops are stored in an array indexed by the loop induction variable
  - Array allocated statically if possible; otherwise dynamically realloc’ed
Case Study: Read Sum

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);
}

define double @sum(double* %x)

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

%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body

%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, for.body

%mul = %0 * %call
%add = %mul + %total
%i.next = %i + 1
%exitcond = %i.next == 10
br %exitcond, for.cleanup, 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

%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

%i = phi [ 0, entry ], [ %i.next, 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.body

for.cleanup
Case Study: Read Sum

define double @sum(double* %x)

for.body

%entry = 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

Active Variables
Case Study: Read Sum

Each register in the for loop represents a distinct active variable every iteration.
Allocate & zero shadow memory per active value
define double @diffe_sum(double* %x, double* %xp)

Cache forward pass variables for use in reverse

for.body

for.cleanup

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

%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

%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
Case Study: Read Sum

```c
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

```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();
}
```
Experimental Setup

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

Enzyme is \textbf{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)
```

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

// diffe_dupnoneed specifies not recomputing the output
void diffef(float* inp, float* d_inp, size_t n, float* d_out) {
    __enzyme_autodiff(f, diffe_dup, inp, d_inp, n, diffe_dupnoneed, (float*)0, d_out);
}
```
Automatic Differentiation & GPUs

• Prior work has not explored reverse mode AD of existing GPU kernels
  • Reversing parallel control flow can lead to incorrect results
  • Complex performance characteristics make it difficult to synthesize efficient code
  • Resource limitations can prevent kernels from running at all
Challenges of Parallel AD

• The adjoint of an instruction increments the derivative of its input

• 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;
}

double gradient_set(double* ar, double* d_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;
}
```
GPU Memory Hierarchy

Per Thread
- Register: ~Bytes
- Use Limits Parallelism

Per Block
- Shared Memory: ~KBs
- Use Limits Parallelism

Per GPU
- Global Memory: ~GBs
- Slower, larger amount of memory
Correct and Efficient Derivative Accumulation

Thread-local memory

- Non-atomic load/store

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

Same memory location across all threads (some shared mem)

- Parallel Reduction

```c
// Same var for all threads
double y;
...
reduce_add(&d_y, val);
```

Others [always legal fallback]

- Atomic increment

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

Slower
Synchronization Primitives

- Synchronization (`sync_threads`) ensures all threads finish executing `codeA` before executing `codeB`.
- Sync is only necessary if A and B may access to the same memory.
- Assuming the original program is race-free, performing a sync at the corresponding location in the reverse ensures correctness.
- Prove correctness of algorithm by cases.
Case 1: Store, Sync, Load

Correct

• Load of d_ptr must happen after all atomicAdds have completed
CUDA Example

```c
__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 diffe_inner(float* a, float* da, float* x, float* dx, float* y, float* dy) {
    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; }
}```
Efficient GPU Code

- Without optimization, GPU gradients must cache a large number of values
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing LLVM optimizations can reduce the overhead
- Unlike the CPU, existing LLVM optimizations aren’t sufficient
- Novel GPU and AD-specific optimizations can speed up by several orders of magnitude
GPU Gradient Overhead

- Evaluation of both original code and gradient
- DG: Discontinuous-Galerkin 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)
Ablation Analysis of Optimizations

Forward (1x) | 10x | 100x | 1000x | OOM
---|---|---|---|---
DG (ROCm) | | | | 5.4x
DG (CUDA) | | | | 17.8x
DG (CUDA) | | | | 116.6x
LBM | Unrolling | MallocCoalescing | PreOptimization
SpecPHI | | | 6.4x 8.7x 19.87x
LULESH | | | 2.0x 2.4x
RSBench | CacheLICM | PreOpt | 4.7x 9.5x
XSBench | Templating PHI LoopBound | PreOpt
Templating PHI LoopBound
Forward (1x) | 3.2x 9.5x 16.3x 25.9x
Overhead above Forward Pass

Unrolling | MallocCoalescing | PreOptimization
Ablation Analysis of Optimizations

DG (ROCm)
- DG (CUDA)
- LBM
- LULESH
- RSbench
- XSBench

Forward (1x) 10x 100x 1000x OOM

Overhead above Forward Pass

Unrolling
Unrolling
MallocCoalescing
InlineCacheABI
PreOptimization
PreOptimization
PreOptimization
PreOptimization
PreOptimization

Allocator Recompute
6.4× 8.7× 19.87×

SpecPHI
2.0× 2.4×

CacheLICM
4.7× 9.5×

Templating
3.2× 9.5× 16.3× 25.9×

OOM
Ablation Analysis of Optimizations

Forward (1x) 10x 100x 1000x OOM

DG (ROCm) 5.4× Unrolling 17.8× MallocCoalescing 1378.1× PreOptimization

DG (CUDA) 116.6× Unrolling 1378.3× PreOptimization

LBM Alloc Recompute InlineCacheABI

LULESH SpecPHI PreOpt 2979.1×

RSBench CacheLICM Inlining PreOpt 6372.2×

XSbench Templating PHI LoopBound PreOpt

Overhead above Forward Pass

Unrolling

PreOptimization
Ablation Analysis of Optimizations

GPU AD is Intractable Without Optimization!
Scalability Analysis (Fixed Work Per Thread)

- Discontinuous Galerkin (Julia & CUDA)
- Discontinuous Galerkin (Julia & ROCm)
- LULESH (C++ & CUDA)
- RSBench (C & CUDA)
- XSBench (C & CUDA)
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
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.
• 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
Scalability Analysis (Fixed Thread Count)

![Graph showing AD Overhead (factor) vs Iterations]

- LBM – Parboil (C & CUDA)
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>
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
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
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
CUDA Automatic Differentiation

- 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.
GPU Automatic Differentiation

• Prior work has not explored reverse mode AD of GPU kernels

• 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
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](http://enzyme.mit.edu) / [github.com/wsmoses/Enzyme](https://github.com/wsmoses/Enzyme))
- PyTorch-Enzyme & TensorFlow-Enzyme imports foreign code in ML workflow
One can specify custom forward/reverse passes of functions by attaching metadata

```
__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
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]
Enzyme Differentiation Algorithm

- Type Analysis
- Activity Analysis
- Synthesize derivatives
  - Forward pass that mirrors original code
  - Reverse pass inverts instructions in forward pass (adjoints) to compute derivatives
- Optimize
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
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\[^1\]

\[^1\] Michael Innes. Don’t Unroll Adjoint: Differentiating SSA-Form Programs. arXiv preprint arXiv:1810.07951, 2018
Differentiation Is Key To Machine Learning

- Hinders application of ML to new domains
- Synthesizing gradients aims to close this gap

// 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;
    }
}

// PyTorch rewrite of nbody simulator
import torch
def step(bodies, dt):
    acc = []
    for i in range(len(bodies)):
        acc.push(torch.zeros([3]))
    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
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

```ml
%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
Case 2: Load, Sync, Store

```c
 codeA(); // load %ptr
 sync_threads;
 codeB(); // store %ptr
 ...
 diffé_codeB(); // load %d_ptr
    // store %d_ptr = 0
 sync_threads;
 diffé_codeA(); // atomicAdd %d_ptr
```

Correct

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

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

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