# Efficient Cross-Platform Automatic Differentiation

William S. Moses

Supertech Presentations
May 20, 2019

# Premise

- “For this reason 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, even though these can all be expressed as SSA.”

  -Michael J Innes, Julia Computing

- That premise is wrong!

# Compilers can do more than compile

- Use LLVM compiler to rewrite program, generating a derivative function from an existing regular function.

```
double square(double x) {
    return x * x;
}
```

```
double dsquare(double x) {
    return __builtin_autodiff(square, x);
}

double square(double x) {
    return x * x;
}

double dsquare(double x) {
    return 2 * x;
}
```

# Compilers can do more than compile

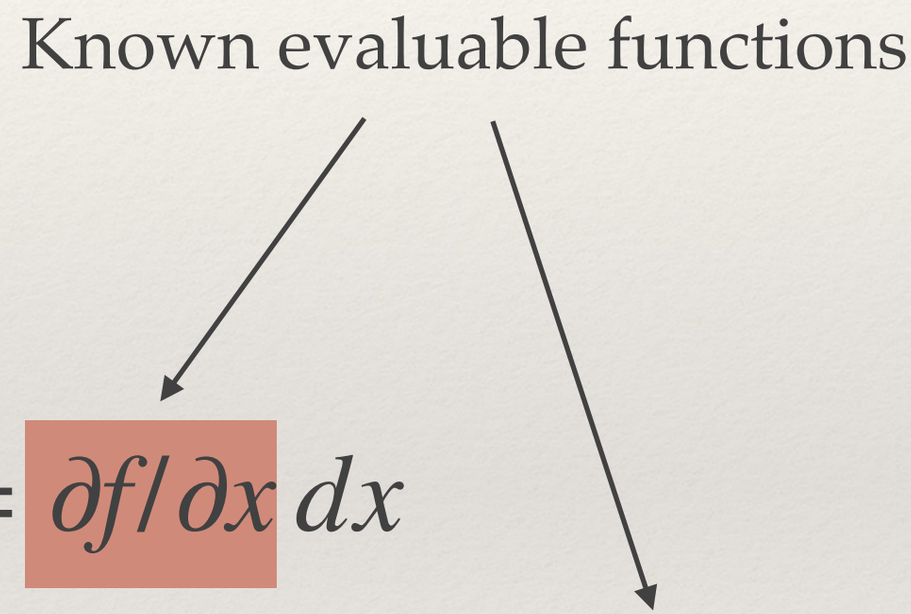
```
; Function Attrs: norecurse nounwind readnone uwtable
define dso_local double @square(double %x) local_unnamed_addr #0 {
entry:
  %mul = fmul fast double %x, %x
  ret double %mul
}

; Function Attrs: nounwind uwtable
define dso_local double @dsquare(double %x) local_unnamed_addr #1 {
entry:
  %0 = tail call double (double (double)*, ...)
@llvm.autodiff.p0f_f64f64f(double (double)* nonnull @square, double %x)
  ret double %0
}

; Function Attrs: norecurse nounwind readnone uwtable
define dso_local double @dsquare(double %x) local_unnamed_addr #0 {
entry:
  %factor.i = fmul fast double %x, 2.000000e+00
  ret double %factor.i
}
```

# Algorithm

- Two types of automatic differentiation:
  - Forward propagates differential input to outputs
    - Ex  y, z, q = f(x)
      - Desire  dy/dx, dz/dx, dq/dx
  - Reverse propagates differential output to inputs
    - Ex  y = f(a, b, c)
      - Desire  dy/da, dy/db, dy/dc
- Both doable in practice, chose reverse mode since I tend to use gradient more than finding all outputs derivatives

# Forward Mode

- Want $dz/dx$
- $y = f(x)$
- $z = g(y, x)$




- Chain rule: $dy = \partial f/\partial x \, dx$
- $$dz = \partial g/\partial y \, dy + \partial g/\partial x \, dx$$

# Reverse Mode

- Want dz / dx
- y = f(x)
- z = g(y, x)

$$\frac{d}{dz} = \partial y/\partial z \frac{d}{dy} + \partial x/\partial z \frac{d}{dx}$$

$$\frac{d}{dz} = \partial y/\partial g \frac{d}{dy} + \partial x/\partial g \frac{d}{dx}$$

- $$\partial g/\partial y \frac{d}{dz}\Big|_x = \frac{d}{dy}\Big|_x \qquad \partial g/\partial x \frac{d}{dz}\Big|_y = \frac{d}{dx}\Big|_y$$

$$\partial f/\partial x \frac{d}{dy}\Big|_z = \frac{d}{dx}\Big|_z$$

$$\frac{d}{dx} = \frac{d}{dx}\Big|_y + \frac{d}{dx}\Big|_z$$

# Reverse Mode

- In essence, run program backwards, using chain rule relations to set differential's of original values
- Add to differential of values
- For control flow, remember the path you came, and execute that path going backwards
  - For loops are reversed (unless they're parallel!)

```
; Function Attrs: norecurse nounwind readonly uwtable
define dso_local double @relu(double %x) local_unnamed_addr #1 {
entry:
  %cmp = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp, label %cond.true, label %cond.end

cond.true:                                        ; preds = %entry
  %call = tail call fast double @f(double %x)
  br label %cond.end

cond.end:                                         ; preds = %cond.true, %entry
  %cond = phi double [ %call, %cond.true ], [ 0.000000e+00, %entry ]
  ret double %cond
}
```

```
; Function Attrs: norecurse nounwind readonly uwtable
define dso_local double @drelu(double %x) local_unnamed_addr #1 {
entry:
  %cmp.i = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp.i, label %invertcond.true.i, label %differelu.exit

invertcond.true.i:                                ; preds = %entry
  %0 = tail call fastcc double @diffef() #2
  br label %differelu.exit

differelu.exit:                                   ; preds = %entry, %invertcond.true.i
  %"x'de.0.i" = phi double [ %0, %invertcond.true.i ], [ 0.000000e+00, %entry ]
  ret double %"x'de.0.i"
}
```

# Efficient Implementation

- When running the program backwards, you might need to use some variables that are no longer in scope.
- Ideas!
  - Store all variables (historically on a tape), and lookup when needed
  - Tapes are slow and require a second stack (dynamic add/remove memory). Instead store in static allocation at top of function. This way LLVM's optimization passes can get rid of the extra memory when possible!
  - Compute forward and backwards paths together (to identify reuse)

# Efficient Implementation

- To store all values without dynamic memory, must allocate upper bound of memory usage
  - Only deal with loops of known size (instead of say while loop across linked list nodes)
- Allocate memory for all SSA nodes [even if don't run that path] to ensure availability
  - In theory could use more memory than stack-approach
  - In practice use far less since most memory is optimized out

```
; Function Attrs: norecurse nounwind readonly uwtable
define internal double @differelu(double %x) #1 {
entry:
  %"cond'ret" = alloca double   <-  store double 1.000000e+00, double* %"cond'ret"
  %"call'de" = alloca double    <-  store double 0.000000e+00, double* %"call'de"
  %"x'de" = alloca double       <-  store double 0.000000e+00, double* %"x'de"
  %cmp = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp, label %cond.true, label %cond.end

cond.true:                                        ; preds = %entry
  %call = tail call fast double @f(double %x)
  br label %cond.end

cond.end:                                         ; preds = %cond.true, %entry
  %cond = phi double [ %call, %cond.true ], [ 0.000000e+00, %entry ]
  br label %invertcond.end

invertcond.end:                                   ; preds = %cond.end
  %8 = fcmp fast ogt double %x, 0.000000e+00
  %9 = load double, double* %"call'de"
  %10 = load double, double* %"cond'ret"
  %11 = select i1 %8, double %10, double %9
  store double %11, double* %"call'de"
  store double 0.000000e+00, double* %"cond'ret"
  br i1 %8, label %invertcond.true, label %invertentry

invertcond.true:                                  ; preds = %invertcond.end
  %3 = call double @diffef(double %x)
  %4 = load double, double* %"call'de"
  %5 = fmul fast double %4, %3
  %6 = load double, double* %"x'de"
  %7 = fadd fast double %6, %5
  store double %7, double* %"x'de"
  store double 0.000000e+00, double* %"call'de"
  br label %invertentry

invertentry:                                      ; preds = %invertcond.true, %invertcond.end
  %0 = load double, double* %"x'de"
  ret %0
}
```

# Run LLVM memory optimizations

```
; Function Attrs: norecurse nounwind readonly uwtable
define internal double @differelu(double %x) #1 {
entry:
  %cmp = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp, label %cond.true, label %invertcond.end

cond.true:                                        ; preds = %entry
  %call = tail call fast double @f(double %x)
  br label %invertcond.end

invertcond.end:                                   ; preds = %cond.true, %entry
  %cond = phi double [ %call, %cond.true ], [ 0.000000e+00, %entry ]
  %5 = fcmp fast ogt double %x, 0.000000e+00
  %6 = select i1 %5, double 1.000000e+00, double 0.000000e+00
  br i1 %5, label %invertcond.true, label %invertentry

invertcond.true:                                  ; preds = %invertcond.end
  %2 = call double @diffef(double %x)
  %3 = fmul fast double %6, %2
  %4 = fadd fast double 0.000000e+00, %3
  br label %invertentry

invertentry:                                      ; preds = %invertcond.true, %invertcond.end
  %"x'de.0" = phi double [ %4, %invertcond.true ], [ 0.000000e+00, %invertcond.end ]
  ret double %"x'de.0"
}
```

# Now CSE / propagation

```
; Function Attrs: norecurse nounwind readonly uwtable
define internal double @differelu(double %x) #1 {
entry:
  %cmp = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp, label %cond.true, label %invertcond.end

cond.true:                                        ; preds = %entry
  %call = tail call fast double @f(double %x)
  br label %invertcond.end

invertcond.end:                                   ; preds = %cond.true, %entry
  %cond = phi double [ %call, %cond.true ], [ 0.000000e+00, %entry ]
  br i1 %cmp, label %invertcond.true, label %invertentry

invertcond.true:                                  ; preds = %invertcond.end
  %2 = call double @diffef(double %x)
  br label %invertentry

invertentry:                                      ; preds = %invertcond.true, %invertcond.end
  %"x'de.0" = phi double [ %2, %invertcond.true ], [ 0.000000e+00, %invertcond.end ]
  ret double %"x'de.0"
}
```

# Finally Dead Code Elimination

```
; Function Attrs: norecurse nounwind readonly uwtable
define internal double @differelu(double %x) #1 {
entry:
  %cmp = fcmp fast ogt double %x, 0.000000e+00
  br i1 %cmp, label %invertcond.true, label %invertentry

invertcond.true:                                  ; preds = %entry
  %2 = call double @diffef(double %x)
  br label %invertentry

invertentry:                                      ; preds = %invertcond.true, %entry
  %"x'de.0" = phi double [ %2, %invertcond.true ], [ 0.000000e+00, %entry ]
  ret double %"x'de.0"
}
```

# Essentially the optimal hand-compiled program!

# Advanced Material

- Handle pointers by keeping separate differential structs of memory
- Detection of constants (with respect to input not necessarily compiler constant) to simply derivatives
- Heuristic to consider recompute
- Don’t need to compute forward pass unless necessary!

# Advanced Material

```
double sum(double* x, unsigned long n) {
    double total = 0;
    for(int i=0; i<=n; i++)
       total += x[i];
    return total;
}

void dsum(double* x, double* xp, unsigned long n) {
    __builtin_autodiff(sum, x, xp, n);
}
```

```
define dso_local double @sum(double* nocapture readonly %x, i64 %n) #0 {
entry:
  br label %for.body

for.cond.cleanup:                                 ; preds = %for.body
  %add.lcssa = phi double [ %add, %for.body ]
  ret double %add.lcssa

for.body:                                         ; preds = %for.body, %entry
  %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next, %for.body ]
  %total.07 = phi double [ 0.000000e+00, %entry ], [ %add, %for.body ]
  %arrayidx = getelementptr inbounds double, double* %x, i64 %indvars.iv
  %0 = load double, double* %arrayidx, align 8, !tbaa !2
  %add = fadd fast double %0, %total.07
  %indvars.iv.next = add nuw i64 %indvars.iv, 1
  %exitcond = icmp eq i64 %indvars.iv, %n
  br i1 %exitcond, label %for.cond.cleanup, label %for.body
}
```

Derivative doesn't need to even compute forward pass!

No stack!

Handles memory addresses (primitive here)

```
; Function Attrs: nounwind uwtable
define dso_local void @dsum(double* %x, double* %xp, i64 %n) local_unnamed_addr #1 {
entry:
  br label %invertfor.body.i

invertfor.body.i:                                 ; preds = %invertfor.body.i, %entry
  %"indvars.iv'phi.i" = phi i64 [ %0, %invertfor.body.i ], [ %n, %entry ]
  %0 = sub i64 %"indvars.iv'phi.i", 1
  %"arrayidx'ip.i" = getelementptr double, double* %xp, i64 %"indvars.iv'phi.i"
  %1 = load double, double* %"arrayidx'ip.i"
  %2 = fadd fast double %1, 1.000000e+00
  store double %2, double* %"arrayidx'ip.i"
  %3 = icmp ne i64 %"indvars.iv'phi.i", 0
  br i1 %3, label %invertfor.body.i, label %diffesum.exit

diffesum.exit:                                    ; preds = %invertfor.body.i
  ret void
}
```

```
double
conv_layer(size_t IN, size_t OUT, size_t NUM,
           const double* __restrict W, const double* __restrict b,
           const mnist_image_t* __restrict input,
           const uint8_t* __restrict true_output) {

  double* output = (double*)malloc(sizeof(double)*NUM*OUT);
  double sum = 0;

  for(int n=0; n<NUM; n++)

…
```

Complex data types

Internal allocation / freeing

```
; Function Attrs: noinline nounwind uwtable
define internal void @diffeconv_layer(i64 %IN, i64 %OUT, i64 %NUM, double* noalias nocapture readonly %W, double*
%"W'", double* noalias nocapture readonly %b, double* %"b'", %struct.mnist_image_t_* noalias nocapture readonly
%input, i8* noalias nocapture readonly %true_output) #7 {
entry:
  %0 = add i64 %OUT, -1
  %1 = add i64 %NUM, -1
  %2 = mul i64 %OUT, %NUM
  %mallocsize = mul i64 %2, 8
  %malloccall = tail call i8* @malloc(i64 %mallocsize)
  %sub.us.us_arraycache = bitcast i8* %malloccall to double*
  %cmp106 = icmp eq i64 %NUM, 0
  br i1 %cmp106, label %for.cond.cleanup, label %for.cond3.preheader.lr.ph

for.cond3.preheader.lr.ph:                        ; preds = %entry
  %cmp5103 = icmp eq i64 %OUT, 0
  %cmp15101 = icmp eq i64 %IN, 0
  br i1 %cmp5103, label %for.cond.cleanup, label %for.cond3.preheader.us.preheader

for.cond3.preheader.us.preheader:                 ; preds = %for.cond3.preheader.lr.ph
  %3 = add i64 %IN, -1
  br label %for.cond3.preheader.us
…
```

# Matrix Vector Multiply, sum loss

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

# Matrix Vector Multiply, sum loss

500 x 500

| | ECPA | Adept |
|---|---|---|
| Normal | 0.0004 | 0.0006 |
| Forward | 0.0004 | 0.0418 |
| Forward+Reverse | 0.0010 | SIGSEG |

4000 x 4000

| | ECPA |
|---|---|
| Normal | 0.6857 |
| Forward | 0.6871 |
| Forward+Reverse | 0.0471 |

# Matrix Vector Multiply, square loss

```
double matvec_real(double* mat, double* vec) {
  double *out = (double*)malloc(sizeof(double)*N);
  for(int i=0; i<N; i++) {
    out[i] = 0;
    for(int j=0; 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;
}
```

# Mat Vec, sum square loss

600 x 600

| | ECPA | Adept |
|---|---|---|
| Normal | 0.0004 | 0.0006 |
| Forward | 0.0004 | 0.0418 |
| Forward+Reverse | 0.0010 | SIGSEG |

4000 x 4000

| | ECPA |
|---|---|
| Normal | 0.6857 |
| Forward | 0.6856 |
| Forward+Reverse | 0.8144 |

~10x faster than Tim's last week on comparison of equivalent programs
[super primitive test, take with grain of salt]

# Taylor Expand Log

10000000 iterations

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

```
function jl_f1(f::Float64)
    g = 0 * f
    for i = 1:10000000
        g += f^i / i
    end
    g
end
```

```
# taylor series for -log(1-x)
# eval at -log(1-1/2) = -log(1/2)
# d/dx -log(1-x) = 1/(1-x)
# d/dx fn(x) |x=0.5 = 2

^Makes it very easy to check correctness
```

# Taylor Expand Log

10000000 iterations

| | ECPA | Adept | EPCA-Julia | Zygote-Julia | AutoGrad-julia |
|---|---|---|---|---|---|
| Normal | 3.74 | 3.72 | 3.82 | 3.82 | 3.82 |
| Forward | 3.74 | 4.56 | same | same | same |
| Forward+Reverse | 3.90 | 4.65 | 3.95 | 44.694 | 896.30 |

# Taylor Expand Log

10000000 iterations

| | ECPA | Adept | EPCA-Julia | Zygote-Julia | AutoGrad-julia |
|---|---|---|---|---|---|
| Normal | 3.74 | 3.72 | 3.82 | 3.82 | 3.82 |
| Forward | 3.74 | 4.56 | same | same | same |
| Forward+Reverse | 3.90 | 4.65 | 3.95 | 44.694 | 896.30 |

| BENCHMARK | FORWARD | ZYGOTE | PYTORCH | REVERSEDIFF |
|---|---|---|---|---|
| SINCOS | 15.9NS | 20.7NS | 69,900NS | 670NS |
| LOOP | 4.17$\mu$S | 29.5$\mu$S | 17,500$\mu$S | 171$\mu$S |
| LOGSUMEXP | 0.96$\mu$S | 1.26$\mu$S | 219$\mu$S | 15.9$\mu$S |
| LOGISTIC REGRESSION | 4.67$\mu$S | 17.6$\mu$S | 142$\mu$S | 89.9$\mu$S |
| 2-LAYER MNIST MLP | 27.7$\mu$S | 207$\mu$S | 369$\mu$S | N/A |

# 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

```
double derivative = __builtin_autodiff(logger, x);
```

# Taylor Expand Log

```
derivative = autodiff(jl_f2, Tuple{typeof(jl_f2),Array{Float64}}, x)
```

# Conclusion

- Automatic differentiation can be done on a highly-used cross-platform, with performance ***benefits*** (not slowdowns)
- Autodiff intrinsic usable by anyone who goes to LLVM IR (Tensorflow, Rust, C/C++, Julia, etc) — Frontend for Julia & C at the moment
- Doesn't require modification of existing code "just works"
- Future Work:
  - More stress tests
  - Non "idempotent" readwrite memory
  - Recursive Functions / split forward and reverse passes
  - Heuristic tuning