LAGrad: Leveraging the MLIR Ecosystem for Efficient Differentiable Programming

US LLVM Developers' Meeting
November 9, 2022

Mai Jacob Peng

McGill University

✉ jacobmpeng@gmail.com

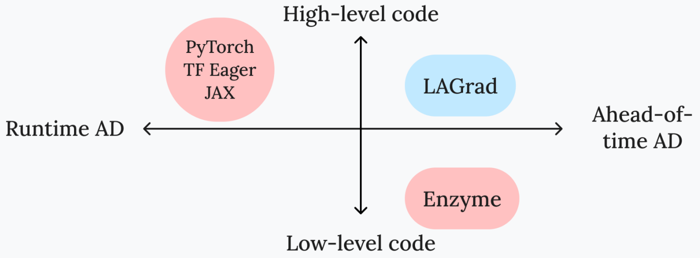
Motivation

- Gradients are everywhere in machine learning, computer vision, etc.
- Virtually all modern deep learning uses gradient-based methods via **Automatic Differentiation** (AD) to train.
- Recent growing attention to **Differentiable Programming**: express models as code, then train via AD.
- We want to make this process more efficient.

Differentiating through a fluid simulation[1]

[1]https://github.com/HIPS/autograd

The Autodiff Landscape



- Runtime AD: can't optimize whole program ahead of time.
- Low-level code: high-level information lost; harder to optimize.

We introduce LAGrad: perform **compile-time** AD in MLIR, then exploit **high-level** information to optimize.

Background: reverse-mode AD

Consider a function $y = f(g(h(x)))$. We would like to compute $\frac{dy}{dx}$.

$$z_1 = h(x) \qquad z_2 = g(z_1) \qquad y = f(z_2)$$

We can apply the chain rule of calculus:

$$\frac{dy}{dx} = \frac{dy}{dz_2}\frac{dz_2}{dz_1}\frac{dz_1}{dx}$$

We break our function into small pieces, differentiate each piece, then recombine.

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Primal

$$z = wx + b$$

$$\sigma = 1 + e^{-z}$$

$$y = \frac{1}{\sigma}$$

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Primal	Adjoint
$z = wx + b$	$\frac{dy}{dy} = 1$ *// seed value*
$\sigma = 1 + e^{-z}$	
$y = \frac{1}{\sigma}$	

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Primal	Adjoint
$z = wx + b$	$\frac{dy}{dy} = 1$ *// seed value*
$\sigma = 1 + e^{-z}$	$\frac{dy}{d\sigma} = (\frac{dy}{dy})\frac{-1}{\sigma^2}$
$y = \frac{1}{\sigma}$	

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Primal	Adjoint
$z = wx + b$	$\frac{dy}{dy} = 1$ *// seed value*
$\sigma = 1 + e^{-z}$	$\frac{dy}{d\sigma} = (\frac{dy}{dy})\frac{-1}{\sigma^2}$
$y = \frac{1}{\sigma}$	$\frac{dy}{dz} = (\frac{dy}{d\sigma})(-e^{-z})$

Example: reverse-mode AD of a function

We want to compute $\frac{dy}{dw}, \frac{dy}{db}$ for the following function:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

Primal

$$z = wx + b$$

$$\sigma = 1 + e^{-z}$$

$$y = \frac{1}{\sigma}$$

Adjoint

$$\frac{dy}{dy} = 1 \quad \textit{// seed value}$$

$$\frac{dy}{d\sigma} = \left(\frac{dy}{dy}\right)\frac{-1}{\sigma^2}$$

$$\frac{dy}{dz} = \left(\frac{dy}{d\sigma}\right)(-e^{-z})$$

$$\frac{dy}{dw} = \frac{dy}{dz}x$$

$$\frac{dy}{db} = \frac{dy}{dz}$$

Multidimensional reverse-mode AD

When our functions have many inputs and outputs, $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$, the derivative generalizes to the **Jacobian**:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Multidimensional reverse-mode AD

When our functions have many inputs and outputs, $\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m$, the derivative generalizes to the **Jacobian**:

$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}, \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{y}} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 1 \end{bmatrix} = \mathbf{I}_m$$

Our fixed seed value becomes the ith column of $\frac{\partial \mathbf{y}}{\partial \mathbf{y}}$:

$$\bar{\mathbf{y}}_i = \frac{\partial \mathbf{y}}{\partial y_i} = \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^\top$$

Note how sparse this is! This is important later on.

The linalg.generic op

The core of the `linalg` dialect is the `generic` op. To represent a dot product:

```
linalg.generic { indexing_maps = [
    (d0) -> (d0)  // map for A (size n)
    (d0) -> (d0)  // map for B (size n)
    (d0) -> ()    // map for C (size 1)
    ]} ins(%A, %B) outs(%C) {
      yield %c + %a * %b : f32
    }
```

This conceptually represents a loop:

```
for d0 from 0 to n:
  C[] += A[d0] * B[d0]
```

The linalg.generic op

Matrix-Vector multiplication:

```
linalg.generic { indexing_maps = [
    (d0, d1) -> (d0, d1)  // map for A (m by n)
    (d0, d1) -> (d1)      // map for B (size n)
    (d0, d1) -> (d0)      // map for C (size m)
    ]} ins(%A, %B) outs(%C) {
      yield %c + %a * %b : f32
    }
```

Which represents:

```
for d0 from 0 to m:
  for d1 from 0 to n:
    C[d0] += A[d0, d1] * B[d1]
```

The linalg.generic op

Matrix-Matrix multiplication:

```
linalg.generic { indexing_maps = [
    (d0, d1, d2) -> (d0, d2) // map for A (m by k)
    (d0, d1, d2) -> (d2, d1) // map for B (k by n)
    (d0, d1, d2) -> (d0, d1) // map for C (m by n)
    ]} ins(%A, %B) outs(%C) {
      yield %c + %a * %b : f32
    }
```

Which represents:

```
for d0 from 0 to m:
  for d1 from 0 to n:
    for d2 from 0 to k:
      C[d0, d1] += A[d0, d2] * B[d2, d1]
```

Adjoint Sparsity

Recall: when computing full Jacobians, we fix the output gradient to a one-hot seed vector (a column of the identity matrix).

$$\bar{\mathbf{y}}_i \in \mathbb{R}^m = \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^\top$$

This naturally leads to many differentiated ops that produce sparse tensors (**per dimension**) with highly regular patterns. For instance:

$$\mathbf{C} = \mathbf{AB}$$

$$\mathbf{A} = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} & & \\ & \bullet & \\ & & \end{bmatrix} \implies \mathbf{C} = \begin{bmatrix} & \bullet & \\ & \bullet & \\ & \bullet & \end{bmatrix}$$

Adjoint Sparsity

Recall: when computing full Jacobians, we fix the output gradient to a one-hot seed vector (a column of the identity matrix).

$$\overline{\mathbf{y}}_i \in \mathbb{R}^m = \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix}^\top$$

This naturally leads to many differentiated ops that produce sparse tensors (**per dimension**) with highly regular patterns. For instance:

$$\mathbf{C} = \mathbf{AB}$$

$$\mathbf{A} = \begin{array}{c|c} & \text{dense} \\ \hline \text{dense} & \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix} \end{array}, \quad \mathbf{B} = \begin{array}{c|c} & \text{sparse} \\ \hline \text{sparse} & \begin{bmatrix} & & \\ & \bullet & \\ & & \end{bmatrix} \end{array} \implies \mathbf{C} = \begin{array}{c|c} & \text{sparse} \\ \hline \text{dense} & \begin{bmatrix} & \bullet & \\ & \bullet & \\ & \bullet & \end{bmatrix} \end{array}$$

Sparse Propagation

These sparsity patterns are easily predicted when used in `linalg` ops:

```
linalg.generic { indexing_maps = [
  (d0, d1, d2) -> (d0, d2) // for %A
  (d0, d1, d2) -> (d2, d1) // for %B
  (d0, d1, d2) -> (d0, d1) // for %C
]} ins(%A, %B) outs(%C) ...
```

We want to predict the sparsity of **C**:

- **B** is sparse along both dims: we mark `d2` and `d1` sparse
- Look at the map for **C**: {`d0, d1`}
- Take the intersection: {`d0, d1`} $\cap$ {`d2, d1`} $=$ {`d1`}

The compiler then generates code to exploit this sparsity.

Sparse Propagation

These sparsity patterns are easily predicted when used in `linalg` ops:

$$\mathbf{A} = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} & & \\ & \bullet & \\ & & \end{bmatrix} \implies \mathbf{C} = \begin{bmatrix} & \bullet & \\ & \bullet & \\ & \bullet & \end{bmatrix}$$

We want to predict the sparsity of **C**:

- **B** is sparse along both dims: we mark `d2` and `d1` sparse
- Look at the map for **C**: `{d0, d1}`
- Take the intersection: `{d0, d1}` $\cap$ `{d2, d1}` $=$ `{d1}`

The compiler then generates code to exploit this sparsity.

AD over loops

Consider the function $y = \sum_{i=0}^{n} xi^2$. We can express this with a loop:

```
y = 0.0f
for i from 0 to n:
  z = i * i
  y += z * x
```

Now, to compute $\frac{dy}{dx}$. How to deal with the loop? One strategy is to fully unroll:

```
z0 = 0 * 0
y += z0 * x
z1 = 1 * 1
y += z1 * x
...
zn = (n - 1) * (n - 1)
y += zn * x
```

$\xrightarrow{AD}$

```
dy = 1.0f
dx += dy * zn
dx += dy * zn_minus_1
...
dx += dy * z2
dx += dy * z1
dx += dy * z0
```

AD over loops

Consider the function $y = \sum_{i=0}^{n} xi^2$. We can express this with a loop:

```
y = 0.0f
for i from 0 to n:
  z = i * i
  y += z * x
```

Now, to compute $\frac{dy}{dx}$. How to deal with the loop? One strategy is to fully unroll:

```
z0 = 0 * 0
y += z0 * x
z1 = 1 * 1
y += z1 * x
...
zn = (n - 1) * (n - 1)
y += zn * x
```

$\xrightarrow{AD}$

```
dy = 1.0f
dx += dy * zn
dx += dy * zn_minus_1
...
dx += dy * z2
dx += dy * z1
dx += dy * z0
```

Can we preserve the loop?

```
dy = 1.0f
dx += dy * zn
dx += dy * zn_minus_1
...
dx += dy * z2
dx += dy * z1
dx += dy * d0
```

$\xrightarrow{\text{Re-roll}}$

```
for i from 0 to n:
  z = i * i
  y += z * x

for i from n to 0:
  // Incorrect; z has been overwritten
  dx += dy * z
```

Can we preserve the loop?

```
dy = 1.0f
dx += dy * zn
dx += dy * zn_minus_1
...
dx += dy * z2
dx += dy * z1
dx += dy * d0
```

$\xrightarrow{\text{Re-roll}}$

```
for i from 0 to n:
  z = i * i
  y += z * x

for i from n to 0:
  // Incorrect; z has been overwritten
  dx += dy * z
```

Solution: the **gradient tape**, store required values in memory:

```
tape = allocate array of size n
for i from 0 to n:
  z = i * i
  tape[i] = z

for i from n to 0:
  z = tape[i]
  dx += dy * z
```

Can we preserve the loop?

```
dy = 1.0f
dx += dy * zn
dx += dy * zn_minus_1
...
dx += dy * z2
dx += dy * z1
dx += dy * d0
```

$\xrightarrow{\text{Re-roll}}$

```
for i from 0 to n:
  z = i * i
  y += z * x

for i from n to 0:
  // Incorrect; z has been overwritten
  dx += dy * z
```

Solution: the **gradient tape**, store required values in memory:

```
tape = allocate array of size n
for i from 0 to n:
  z = i * i
  tape[i] = z

for i from n to 0:
  z = tape[i]
  dx += dy * z
```

Problem: Original program takes $\mathcal{O}(1)$ memory, but adjoint takes $\mathcal{O}(n)$.

Can we avoid the memory overhead?

```
tape = allocate array of size n
for i from 0 to n:
    z = i * i
    tape[i] = z

for i from n to 0:
    z = tape[i]
    dx += dy * z
```

$\Longrightarrow$

```
for i from n to 0:
    z = i * i
    dx += dy * z
```

Can we avoid the memory overhead?

```
tape = allocate array of size n
for i from 0 to n:
  z = i * i
  tape[i] = z

for i from n to 0:
  z = tape[i]
  dx += dy * z
```

$\Longrightarrow$

```
for i from n to 0:
  z = i * i
  dx += dy * z
```

- No tape required! We recompute instead of cache.
- Recomputation is cheap.
- The compiler can trivially remove the primal loop.

Another example

Now consider the function $y = x^n$:

```
p = 1.0f
for i from 0 to n:
  p = p * x
```

- Note that p is carried through the loop (and *overwritten*). At iteration i, it depends on the p values from iterations $1, \ldots, i-1$.

Can we recompute p?

```
tape = allocate array of size n
p = 1.0f
for i = 0 to n:
  tape[i] = p
  p = p * x

dp = 1.0f, dx = 0.0f
for i = n to 0:
  p = tape[i]
  dx += dp * p
  dp = dp * x
```

$$i = n-1, \quad p = x \times x \times \ldots \times x \times x$$

$$i = n-2, \quad p = x \times x \times \ldots \times x$$

$$\vdots$$

$$i = 2, \quad p = x \times x \times x$$

$$i = 1, \quad p = x \times x$$

$$i = 0, \quad p = x$$

Can we recompute p?

```
tape = allocate array of size n
p = 1.0f
for i = 0 to n:
  tape[i] = p
  p = p * x

dp = 1.0f, dx = 0.0f
for i = n to 0:
  p = tape[i]
  dx += dp * p
  dp = dp * x
```

$$
\begin{aligned}
i &= n-1, & p &= x \times x \times \ldots \times x \times x \\
i &= n-2, & p &= x \times x \times \ldots \times x \\
& & &\vdots \\
i &= 2, & p &= x \times x \times x \\
i &= 1, & p &= x \times x \\
i &= 0, & p &= x
\end{aligned}
$$

If we recompute p, every adjoint iteration needs restart from scratch. Computation goes from $\mathcal{O}(n)$ to $\mathcal{O}(n^2)$.

Tape Size Reduction

When is it beneficial to inline? We can formalize:

- *AdjU*: Set of primal values required to compute the adjoint.
- *IterVals*: Set of loop-carried primal values + values that depend on them.

If $\mathit{AdjU} \cap \mathit{IterVals} = \emptyset$, the required primal values can be computed without depending on previous iterations.

LAGrad uses this heuristic (among others) to avoid emitting a tape during AD.

Tape Size Reduction in Tensor (Pseudo-)MLIR

```
scf.for iv = 0 to n iter_args(r_it = 0.0f) {
  z = iv * iv : f32
  scf.yield r_it + (z * x) : f32
}
```

$IterVals = \{\texttt{r\_it}\} \qquad AdjU = \{\texttt{z, x}\}$

$IterVals \cap AdjU = \emptyset$

Tape Size Reduction in Tensor (Pseudo-)MLIR

```
scf.for iv = 0 to n iter_args(r_it = 0.0f) {
  z = iv * iv : f32
  scf.yield r_it + (z * x) : f32
}
```

$\mathit{IterVals} = \{\texttt{r\_it}\} \qquad \mathit{AdjU} = \{\texttt{z, x}\}$

$\mathit{IterVals} \cap \mathit{AdjU} = \emptyset$

```
scf.for iv = 0 to n iter_args(p_it = 1.0f) {
  p_next = arith.mulf p_it, x : f32
  scf.yield p_next : f32
}
```

$\mathit{IterVals} = \{\texttt{p\_it, p\_next}\} \qquad \mathit{AdjU} = \{\texttt{p\_it, x}\}$

$\mathit{IterVals} \cap \mathit{AdjU} = \{\texttt{p\_it}\} \neq \emptyset$

Tape Size Reduction in Tensor (Pseudo-)MLIR

```
scf.for iv = 0 to n iter_args(r_it = 0.0f) {
  z = iv * iv : f32
  scf.yield r_it + (z * x) : f32
}
```

$\mathit{IterVals} = \{\texttt{r\_it}\} \qquad \mathit{AdjU} = \{\texttt{z, x}\}$

$\mathit{IterVals} \cap \mathit{AdjU} = \emptyset$

```
scf.for iv = 0 to n iter_args(p_it = 1.0f) {
  p_next = arith.mulf p_it, x : f32
  scf.yield p_next : f32
}
```

$\mathit{IterVals} = \{\texttt{p\_it, p\_next}\} \qquad \mathit{AdjU} = \{\texttt{p\_it, x}\}$

$\mathit{IterVals} \cap \mathit{AdjU} = \{\texttt{p\_it}\} \neq \emptyset$

MLIR semantics help us a ton:

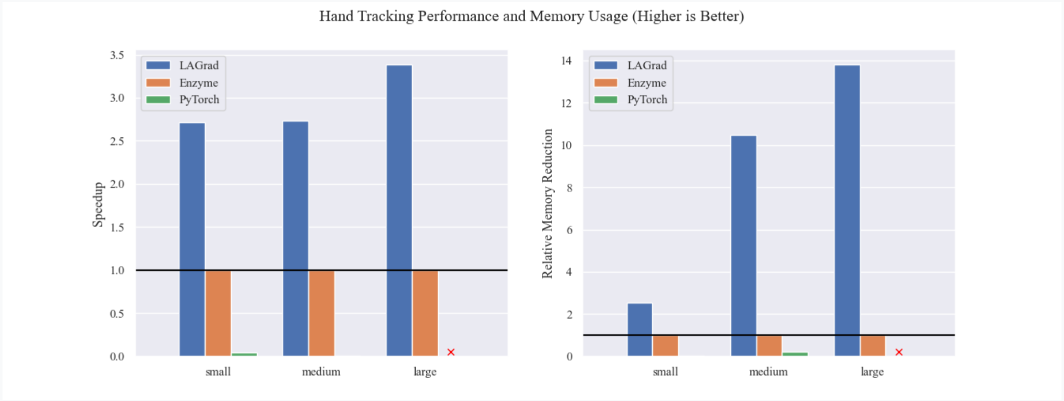
- `iter_args` are explicit
- No worrying about memory side-effects
- Analysis scales to complex programs (nested loops, linalg ops, etc.)

Evaluation Methodology

- We compare run time and memory consumption against Enzyme (state of the art), and PyTorch (popular industry standard) on a standard AD benchmark suite[2].
- The benchmarks are implemented in MLIR using the `linalg`, `tensor`, and `scf` dialects. Enzyme and LAGrad start from the same MLIR.
- We also add a benchmark (2-layer multi-layer perceptron) that leverages optimizations not covered in this talk.

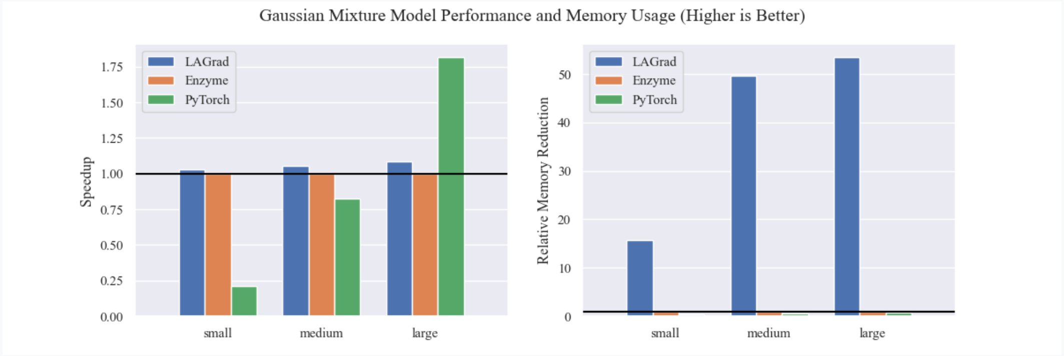
[2] https://github.com/microsoft/ADBench

Evaluation: Adjoint Sparsity



Geomeans: 2.8× speedup over Enzyme while using 8× less memory. 169× speedup over PyTorch while using 61× less memory.

Evaluation: Tape Size Reduction



Geomeans: 74× less memory than PyTorch, 35× less memory than Enzyme without compromising performance.

Summary: Geometric means for all measured datasets

Benchmark	Speedup w.r.t. Enzyme	Memory usage reduction w.r.t. Enzyme	Speedup w.r.t PyTorch	Memory usage reduction w.r.t. PyTorch
GMM	1.1	35.0	6.4	74.0
BA	2.1	0.9	1419.1	103.1
Hand	2.8	7.8	168.7	61.0
LSTM	1.5	3.9	268.6	38.6
MLP	79.6	8.0	2.3	8.8
Geomean	**3.8**	**6.0**	**62.4**	**43.6**

Thanks for listening!

Thank you to my supervisor, Christophe Dubach, and the rest of the Compiler & Accelerator Synthesis lab at McGill.

https://github.com/pengmai

jacobmpeng@gmail.com

Further Reading

- How we compute AdjU sets: https://doi.org/10.1016/j.future.2004.11.009
- Enzyme: https://enzyme.mit.edu/