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.

---

1https://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 \( \frac{dy}{dx} \).

\[
z_1 = h(x) \quad z_2 = g(z_1) \quad y = f(z_2)
\]

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**

\[
z = wx + b
\]

\[
\sigma = 1 + e^{-z}
\]

\[
y = \frac{1}{\sigma}
\]

**Adjoint**

\[
\frac{dy}{dy} = 1 \quad // \text{seed value}
\]

\[
\frac{dy}{dz} = (\frac{dy}{d\sigma})(-e^{-z})
\]
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**

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

- $\frac{dy}{dy} = 1$ // seed value
- $\frac{dy}{d\sigma} = \left(\frac{dy}{dy}\right)^{-1} \sigma^{-2}$
- $\frac{dy}{dz} = \left(\frac{dy}{d\sigma}\right)(-e^{-z})$
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**

- $\frac{dy}{dy} = 1$ // seed value
- $\frac{dy}{d\sigma} = (\frac{dy}{dy})^{-1} \sigma^2$
- $\frac{dy}{dz} = (\frac{dy}{d\sigma})(-e^{-z})$

- $\frac{dy}{dw} = \frac{dy}{dz}x$
- $\frac{dy}{db} = \frac{dy}{dz}$
Multidimensional reverse-mode AD

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

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

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

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

$$
\mathbf{y}_i = \frac{\partial \mathbf{y}}{\partial y_i} = \begin{bmatrix}
\circ & \cdots & \circ & 1 & \circ & \cdots & \circ
\end{bmatrix}^T
$$

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:

```python
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:

```python
for d0 from 0 to n:
    C[] += A[d0] * B[d0]
```
The linalg.generic op

Matrix-Vector multiplication:

```
1 linalg.generic { indexing_maps = [
2   (d0, d1) -> (d0, d1) // map for A (m by n)
3   (d0, d1) -> (d1)   // map for B (size n)
4   (d0, d1) -> (d0)   // map for C (size m)
5  ]} ins(%A, %B) outs(%C) {
6      yield %c + %a * %b : f32
7  }
```

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).

$$\vec{y}_i \in \mathbb{R}^m = [0 \ldots 0 \ 1 \ 0 \ldots 0]^\top$$

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

$$C = AB$$

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

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

\[ \bar{y}_i \in \mathbb{R}^m = [\circ \ldots \circ 1 \circ \ldots \circ]^\top \]

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

\[ C = AB \]

\[ A = \begin{bmatrix} \text{dense} & \text{dense} & \cdots & \text{dense} \\ \text{dense} & \cdots & \cdots & \cdots \\ \text{dense} & \cdots & \cdots & \cdots \end{bmatrix}, \quad B = \begin{bmatrix} \text{sparse} \\ \text{sparse} \\ \text{sparse} \end{bmatrix}, \quad \Rightarrow \quad C = \begin{bmatrix} \text{sparse} \\ \text{sparse} \end{bmatrix} \]
Sparse Propagation

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

```python
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 \(d_2\) and \(d_1\) sparse
- Look at the map for **C**: \(\{d_0, d_1\}\)
- Take the intersection: \(\{d_0, d_1\} \cap \{d_2, d_1\} = \{d_1\}\)

The compiler then generates code to exploit this sparsity.
Sparse Propagation

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

\[
A = \begin{bmatrix}
  \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot \\
  \cdot & \cdot & \cdot & \cdot \\
\end{bmatrix}, \quad B = \begin{bmatrix}
  \cdot \\
\end{bmatrix} \quad \Rightarrow \quad C = \begin{bmatrix}
  \cdot \\
  \cdot \\
\end{bmatrix}
\]

We want to predict the sparsity of \(C\):

- \(B\) is sparse along both dims: we mark \(d_2\) and \(d_1\) sparse
- Look at the map for \(C\): \(\{d_0, d_1\}\)
- Take the intersection: \(\{d_0, d_1\} \cap \{d_2, d_1\} = \{d_1\}\)

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:

```python
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:

```python
z0 = 0 * 0
y += z0 * x
dy = 1.0f
dx += dy * zn
```

```bash
zn = (n - 1) * (n - 1)
y += zn * x
dx += dy * zn_minus_1
```

...
AD over loops

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

```plaintext
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:

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

\[ dy = 1.0f \]
\[ dx += dy * zn \]
\[ dx += dy * zn_{-1} \]
\[ ... \]
\[ dx += dy * z2 \]
\[ dx += dy * z1 \]
\[ dx += dy * z0 \]
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

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 $O(1)$ memory, but adjoint takes $O(n)$. 
Can we preserve the loop?

\[
\begin{align*}
dy &= 1.0f \\
dx &+= dy \ast zn \\
dx &+= dy \ast zn\_minus\_1 \\
&\vdots \\
dx &+= dy \ast z2 \\
dx &+= dy \ast z1 \\
dx &+= dy \ast d0 \\
end{align*}
\]

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

\[
\begin{align*}
tape &\text{ = allocate array of size n} \\
&\text{for } i \text{ from 0 to } n:\ \\
&\quad z = i \ast i \\
&\quad \text{tape}[i] = z \\
&\text{for } i \text{ from } n \text{ to } 0:\ \\
&\quad z = \text{tape}[i] \\
&\quad dx += dy \ast z
\end{align*}
\]
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

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 $O(1)$ memory, but adjoint takes $O(n)$. 
Can we avoid the memory overhead?

```plaintext
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
⇒
for i from n to 0:
    z = i * i
    dx += dy * z
```
Can we avoid the memory overhead?

```python
1 tape = allocate array of size n
2 for i from 0 to n:
3     z = i * i
4     tape[i] = z
5 for i from n to 0:
6     z = tape[i]
7     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 $y = x^n$:

```python
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 $p$?

```python
# 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
```

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

$$
\begin{align*}
    i = n - 1, & \quad p = x \times x \times \ldots \times x \\
    i = n - 2, & \quad p = x \times x \times \ldots \times x \\
    \vdots & \\
    i = 2, & \quad p = x \times x \times x \\
    i = 1, & \quad p = x \times x \\
    i = 0, & \quad p = x
\end{align*}
$$
Can we recompute $p$?

```plaintext
1 tape = allocate array of size n
2 p = 1.0f
3 for i = 0 to n:
   4 tape[i] = p
   5 p = p * x
6
7 dp = 1.0f, dx = 0.0f
8 for i = n to 0:
9   p = tape[i]
10  dx += dp * p
11  dp = dp * x

i = n − 1, $p = x \times x \times \ldots \times x \times x$

i = n − 2, $p = x \times x \times \ldots \times x$

...$

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

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

i = 0, $p = x$
```

If we recompute $p$, every adjoint iteration needs restart from scratch. Computation goes from $\Theta(n)$ to $\Theta(n^2)$.
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 \( AdjU \cap IterVals = \emptyset \), 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 = \{r_it\} \hspace{1cm} AdjU = \{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
}
```

\[IterVals = \{r_it\} \quad AdjU = \{z, x\}\]

\[IterVals \cap AdjU = \emptyset\]

```
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
}
```

\[IterVals = \{p_it, p_next\} \quad AdjU = \{p_it, x\}\]

\[IterVals \cap AdjU = \{p_it\} \neq \emptyset\]
Tape Size Reduction in Tensor (Pseudo-)MLIR

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

\[ \text{IterVals} = \{r\_it\} \quad \text{AdjU} = \{z, x\} \]

\[ \text{IterVals} \cap \text{AdjU} = \emptyset \]

```mlir
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
}
```

\[ \text{IterVals} = \{p\_it, p\_next\} \quad \text{AdjU} = \{p\_it, x\} \]

\[ \text{IterVals} \cap \text{AdjU} = \{p\_it\} \neq \emptyset \]

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: $2.8 \times$ speedup over Enzyme while using $8 \times$ less memory. $169 \times$ speedup over PyTorch while using $61 \times$ less memory.
Evaluation: Tape Size Reduction

Gaussian Mixture Model Performance and Memory Usage (Higher is Better)

Geomeans: $74 \times$ less memory than PyTorch, $35 \times$ less memory than Enzyme without compromising performance.
## Summary: Geometric means for all measured datasets

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