

# Hackable Autodiff

Extending Enzyme to MLIR for Reverse Mode Gradients

---

Martin Eppert<sup>1</sup> Mai Jacob Peng<sup>2</sup>

Enzyme Conference, February 23, 2023

<sup>1</sup>Technical University of Munich

<sup>2</sup>McGill University

# Motivation

---

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:
  - `linalg`: high-level linear algebra (matmuls, convolution, dot products)

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:
  - `linalg`: high-level linear algebra (matmuls, convolution, dot products)
  - `scf`: structured control flow (if statements, for/while loops)

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:
  - `linalg`: high-level linear algebra (matmuls, convolution, dot products)
  - `scf`: structured control flow (if statements, for/while loops)
  - `llvm`: closely maps to LLVM IR (GEP, ptrtoint, cond\_br)

# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:
  - `linalg`: high-level linear algebra (matmuls, convolution, dot products)
  - `scf`: structured control flow (if statements, for/while loops)
  - `llvm`: closely maps to LLVM IR (GEP, ptrtoint, cond\_br)
  - `memref`: shaped regions of memory (analogous to NumPy's `ndarray`)

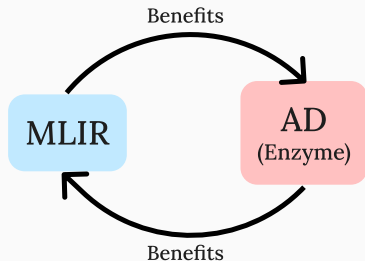


# What is MLIR?

What even is MLIR? Why would we use it?

- *Multi-Level Intermediate Representation*: compiler infrastructure and IR that is **extensible**.
- Grouped into **dialects** with varying abstraction level. Some examples:
  - `linalg`: high-level linear algebra (matmuls, convolution, dot products)
  - `scf`: structured control flow (if statements, for/while loops)
  - `llvm`: closely maps to LLVM IR (GEP, ptrtoint, cond\_br)
  - `memref`: shaped regions of memory (analogous to NumPy's `ndarray`)
  - **Your dialect!** Easy to add new dialects with customizable semantics

## Why use MLIR?



We believe bringing together **MLIR** and **AD** (in the form of Enzyme) will mutually benefit each other.

- AD is ubiquitous in users of MLIR e.g. in Machine Learning and HPC

- AD is ubiquitous in users of MLIR e.g. in Machine Learning and HPC
  - More than just tensor DSLs! Projects like Polygeist and Flang-new.
  - AD is currently re-implemented for each application
  - Many AD implementations have their own quirks and assumptions (e.g. around in-place updates)

- AD is ubiquitous in users of MLIR e.g. in Machine Learning and HPC
  - More than just tensor DSLs! Projects like Polygeist and Flang-new.
  - AD is currently re-implemented for each application
  - Many AD implementations have their own quirks and assumptions (e.g. around in-place updates)

Many MLIR users target LLVM IR. Why not just use LLVM Enzyme?

## Why not just use LLVM Enzyme?

Matrix multiplication in the  
linalg dialect of MLIR:

```
linalg.matmul
  ins(A, B :
    memref<?x?xf32>,
    memref<?x?xf32>)
  outs(C :
    memref<?x?xf32>)
```



## MLIR benefits AD: memcpy, revisited

In LLVM IR/C/C++, given types  $\neq$  actual types (`void *` everywhere!)

```
void *memcpy(void *dest, const void * src, size_t n);
```



## MLIR benefits AD: memcpy, revisited

In LLVM IR/C/C++, given types  $\neq$  actual types (`void *` everywhere!)

```
void *memcpy(void *dest, const void * src, size_t n);
```

Enzyme requires **type analysis** to address this - can significantly increase compile times

## MLIR benefits AD: memcpy, revisited

In LLVM IR/C/C++, given types  $\neq$  actual types (`void *` everywhere!)

```
void *memcpy(void *dest, const void * src, size_t n);
```

Enzyme requires **type analysis** to address this - can significantly increase compile times

In MLIR, the *canonical* way of copying a MemRef is:

```
memref.copy(%src, %dst) : memref<?xf32>
```

# MLIR benefits AD: memcpy, revisited

In LLVM IR/C/C++, given types  $\neq$  actual types (`void *` everywhere!)

```
void *memcpy(void *dest, const void * src, size_t n);
```

Enzyme requires **type analysis** to address this - can significantly increase compile times

In MLIR, the *canonical* way of copying a MemRef is:

```
memref.copy(%src, %dst) : memref<?xf32>
```

Using MLIR has potential to improve compile times - reduced need for static analysis: **better information in the IR**

- AD in MLIR operates on smaller code size with better information than LLVM

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.

## MLIR benefits AD

- AD in MLIR operates on smaller code size with better information than LLVM
- High level languages are easier to differentiate than low level languages

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.

## MLIR benefits AD

- AD in MLIR operates on smaller code size with better information than LLVM
- High level languages are easier to differentiate than low level languages
- Some optimizations easier to express on high-level code<sup>1</sup>

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.

- AD in MLIR operates on smaller code size with better information than LLVM
- High level languages are easier to differentiate than low level languages
- Some optimizations easier to express on high-level code<sup>1</sup>
  - Enzyme-MLIR allows reuse of optimization passes specific to AD for different applications

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.

- AD in MLIR operates on smaller code size with better information than LLVM
- High level languages are easier to differentiate than low level languages
- Some optimizations easier to express on high-level code<sup>1</sup>
  - Enzyme-MLIR allows reuse of optimization passes specific to AD for different applications
  - Core thesis of Enzyme: **optimize your code before running AD**. This also applies to MLIR!

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.



# MLIR benefits AD

- AD in MLIR operates on smaller code size with better information than LLVM
- High level languages are easier to differentiate than low level languages
- Some optimizations easier to express on high-level code<sup>1</sup>
  - Enzyme-MLIR allows reuse of optimization passes specific to AD for different applications
  - Core thesis of Enzyme: **optimize your code before running AD**. This also applies to MLIR!
- Enzyme-MLIR makes AD **extensible**: add AD to your dialect!

---

<sup>1</sup>Peng and Dubach, 'LAGrad: Statically Optimized Differentiable Programming in MLIR'.

# Implementation

---

Interfaces in MLIR are key in making Enzyme-MLIR **dialect agnostic**

- In general, analyses and transformations can interact with Ops via their Interfaces

Interfaces in MLIR are key in making Enzyme-MLIR **dialect agnostic**

- In general, analyses and transformations can interact with Ops via their Interfaces
- Enzyme-MLIR differentiates any Op that implements `ReverseAutoDiffOpInterface`

Primary algorithm<sup>2</sup>:

1. Initialize/zero out shadow memory and caches
2. Construct reversed control flow graph
3. Synthesize differential operations

---

<sup>2</sup>Moses and Churavy, 'Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients'.

Primary algorithm<sup>2</sup>:

1. Initialize/zero out shadow memory and caches

```
ReverseAutoDiffOpInterface {  
    // Store any values required to compute the  
    // gradient  
    SmallVector<Value> cacheValues(...)  
  
    // Initialize required shadow memory  
    void createShadowValues(...) }
```

2. Construct reversed control flow graph
3. Synthesize differential operations

---

<sup>2</sup>Moses and Churavy, 'Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients'.

Primary algorithm<sup>2</sup>:

1. Initialize/zero out shadow memory and caches
2. Construct reversed control flow graph
  - Enzyme-MLIR does this for you!
3. Synthesize differential operations

---

<sup>2</sup>Moses and Churavy, 'Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients'.

Primary algorithm<sup>2</sup>:

1. Initialize/zero out shadow memory and caches
2. Construct reversed control flow graph
3. **Synthesize differential operations**

```
ReverseAutoDiffOpInterface {  
    // Synthesize differential instructions  
    void createReverseModeAdjoint(...) }
```

- You can call Enzyme-MLIR to recursively differentiate child regions here!

---

<sup>2</sup>Moses and Churavy, 'Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients'.



# AutoDiffOpInterface

```
ReverseAutoDiffOpInterface {  
  // Store any values required to compute the gradient  
  SmallVector<Value> cacheValues(GradientUtils *)  
  
  // Initialize required shadow memory  
  void createShadowValues(OpBuilder &, GradientUtils *)  
  
  // Synthesize differential instructions  
  void createReverseModeAdjoint(OpBuilder &,  
    GradientUtils *, SmallVector<Value> caches)  
}
```

```
%gradient = "enzyme.init"() : () ->  
    !enzyme.Gradient<f64>
```

```
%1 = "enzyme.get"(%gradient) : (!enzyme.Gradient<f64>)  
    -> f64
```

```
"enzyme.set"(%gradient, %2) : (!enzyme.Gradient<f64>,  
    f64) -> ()
```

```
%cache = "enzyme.init"() : () -> !enzyme.Cache<i32>
```

```
"enzyme.push"(%cache, %1) : (!enzyme.Cache<i32>, i32)  
  -> ()
```

```
%2 = "enzyme.pop"(%cache) : (!enzyme.Cache<i32>) -> i32
```

```
%3 = "enzyme.get"(cache) : (!enzyme.Cache<i32>) -> i32
```

```
// initialization  
%dx = enzyme.init : !enzyme.Gradient<f64>  
%cache = enzyme.init : !enzyme.Cache<f64>
```

## General Pattern

```
// initialization
```

```
%dx = enzyme.init : !enzyme.Gradient<f64>
```

```
%cache = enzyme.init : !enzyme.Cache<f64>
```

```
// forward pass
```

```
enzyme.push %cache, %x
```

```
%result = "some.operation"(%x)
```

## General Pattern

```
// initialization
%dx = enzyme.init : !enzyme.Gradient<f64>
%cache = enzyme.init : !enzyme.Cache<f64>

// forward pass
enzyme.push %cache, %x
%result = "some.operation"(%x)

// backward pass
%x_restored = enzyme.pop %cache
%0 = enzyme.get %dx
%1 = "dsome.operation"(%x_restored, %dresult)
enzyme.set %dx, %1 + %0
```

- Code can be lowered arbitrarily before AD

- Code can be lowered arbitrarily before AD
- Code can be optimized before AD



- Code can be lowered arbitrarily before AD
- Code can be optimized before AD
- Post-AD optimizations are **decoupled** from core AD procedure

- Code can be lowered arbitrarily before AD
- Code can be optimized before AD
- Post-AD optimizations are **decoupled** from core AD procedure
  - **High-level info preserved:** `enzyme.get` vs `memref.load`

- Code can be lowered arbitrarily before AD
- Code can be optimized before AD
- Post-AD optimizations are **decoupled** from core AD procedure
  - **High-level info preserved:** `enzyme.get` vs `memref.load`
  - You can specifically target these in your optimizations!

## Usage Example

---

## Usage Example

Recall: Enzyme-MLIR requires that active ops implement the `ReverseAutoDiffOpInterface`

Consists of three interface methods:

```
// Store any values required to compute the derivative  
SmallVector<Value> cacheValues(GradientUtils *)
```

```
// Initialize required shadow memory  
void createShadowValues(OpBuilder &, GradientUtils *)
```

```
// Synthesize differential instructions  
void createReverseModeAdjoint(OpBuilder &,  
    GradientUtils *, SmallVector<Value> caches)
```

Derivative of  $r = a + b \implies (\partial a = \partial r, \partial b = \partial r)$

Derivative of  $r = a + b \implies (\partial a = \partial r, \partial b = \partial r)$

```
// No caches required
```

```
SmallVector<Value> cacheValues(...) {  
    return SmallVector<Value>();  
}
```

Derivative of  $r = a + b \implies (\partial a = \partial r, \partial b = \partial r)$

```
// No caches required
```

```
SmallVector<Value> cacheValues(...) {  
    return SmallVector<Value>();  
}
```

```
// No shadow required
```

```
void createShadowValues(...) {}
```



Derivative of  $r = a + b \implies (\partial a = \partial r, \partial b = \partial r)$

```
// No caches required
```

```
SmallVector<Value> cacheValues(...) {  
    return SmallVector<Value>();  
}
```

```
// No shadow required
```

```
void createShadowValues(...) {}
```

```
void createReverseModeAdjoint(addOp, builder, gutils,  
    caches) {
```

```
    Value dr = invert(addOp.getResult());
```

```
    // emit:
```

```
    // invert(addOp.LHS) += dr
```

```
    // invert(addOp.RHS) += dr
```

```
}
```

Derivative of  $r = a \times b \implies (\partial a = b \times \partial r, \partial b = a \times \partial r)$

Derivative of  $r = a \times b \implies (\partial a = b \times \partial r, \partial b = a \times \partial r)$

```
SmallVector<Value> cacheValues(...) {  
    Value cachedLHS = // init cache, push LHS  
    Value cachedRHS = // init cache, push RHS  
    return SmallVector<Value>{cachedRHS, cachedLHS};  
}
```

Derivative of  $r = a \times b \implies (\partial a = b \times \partial r, \partial b = a \times \partial r)$

```
SmallVector<Value> cacheValues(...) {  
    Value cachedLHS = // init cache, push LHS  
    Value cachedRHS = // init cache, push RHS  
    return SmallVector<Value>{cachedRHS, cachedLHS};  
}  
  
// No shadow required
```

Derivative of  $r = a \times b \implies (\partial a = b \times \partial r, \partial b = a \times \partial r)$

```
SmallVector<Value> cacheValues(...) {  
    Value cachedLHS = // init cache, push LHS  
    Value cachedRHS = // init cache, push RHS  
    return SmallVector<Value>{cachedRHS, cachedLHS};  
}
```

```
// No shadow required
```

```
void createReverseModeAdjoint(mulOp, ...) {  
    Value dr = invert(mulOp.getResult());  
    // emit:  
    // invert(mulOp.LHS) += cachedRHS * dr  
    // invert(mulOp.RHS) += cachedLHS * dr  
}
```

```
// No caches required
```

```
// No caches required

void createShadowValues(allocOp, ...) {
    // Allocate a shadow MemRef of the same shape/type
    // as the original
    auto shadow = create<memref::AllocOp>(
        allocOp.getType(), allocOp.getShape());
    fillWithZeros(shadow);
    mapShadow(allocOp, shadow);
}

// No synthesis required
void createReverseModeAdjoint(...) {}
```

How do we differentiate `memref.subview`?



How do we differentiate `memref.subview`?

$$A : \text{memref}\langle 2 \times 3 \rangle = \begin{bmatrix} 1 & 0 & 1 \\ 3 & 4 & 1 \end{bmatrix}$$

$$B : \text{memref}\langle 2 \rangle = A[:, 1]$$

How do we differentiate `memref.subview`?

$$A : \text{memref}\langle 2 \times 3 \rangle = \begin{bmatrix} 1 & 0 & 1 \\ 3 & 4 & 1 \end{bmatrix} \quad B : \text{memref}\langle 2 \rangle = A[:, 1]$$

$$\partial A : \text{memref}\langle 2 \times 3 \rangle = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \partial B : \text{memref}\langle 2 \rangle = \partial A[:, 1]$$

Shadow of the subview = Subview of the shadow

```
// No caches required

void createShadowValues(subviewOp, ...) {
    // Shadow of the subview = subview of the shadow
    Value source = invert(subviewOp.getSource());
    auto shadowView = create<memref::SubViewOp>(source,
        subviewOp.getIndices());
    mapShadow(subviewOp, shadowView);
}

// No synthesis required
```

## Future Work

---



- Explore dataflow analysis in MLIR to handle type analysis, activity analysis, differential use analysis, etc
  - MLIR currently lacks a mature alias analysis implementation
- **Optimize!** Explore both general and AD-specific optimizations, benchmark both compile-time and run-time performance

# Open Questions

- Can we expose an API that doesn't require totally buying into MLIR? Can users write their AD implementations in e.g. Julia or Rust?
- What information can be inferred? Is the current set of interface methods necessary and/or sufficient?

Thank you for listening!

# Bibliography

-  Moses, William and Valentin Churavy. 'Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients'. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 12472–12485. URL: <https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf>.
-  Peng, Mai Jacob and Christophe Dubach. 'LAGrad: Statically Optimized Differentiable Programming in MLIR'. In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. CC 2023. Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 228–238. DOI: [10.1145/3578360.3580259](https://doi.org/10.1145/3578360.3580259). URL: <https://doi.org/10.1145/3578360.3580259>.