MimIrADe: Automatic Differentiation in MimIR

Marcel Ullrich
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Sebastian Hack
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Roland Leißa
University of Mannheim
Mannheim, Germany

Abstract

Automatic differentiation (AD) is at the core of all machine learning frameworks and has applications in scientific computing as well. Theoretical research on reverse-mode AD focuses on functional, higher-order languages, enabling AD to be formulated as a series of local, concise program rewrites. These theoretical approaches focus on correctness but disregard efficiency. Practical implementations, however, employ mutation and taping techniques to enhance efficiency. This approach, however, necessitates intricate, low-level, and non-local program transformations.

In this work, we introduce MimIrADe, a functionally inspired AD technique implemented within a higher-order, graph-based ("sea of nodes") intermediate representation (IR). Our method consists of a streamlined implementation and incorporates standard optimizations, resulting in an efficient AD system. The higher-order nature of the IR enables us to utilize concise functional AD methods, expressing AD through local rewrites. This locality facilitates modular high-level extensions, such as matrix operations, in a straightforward manner. Additionally, the graph-based structure of the IR ensures that critical implementation aspects—particularly the handling of shared pullback invocations—are managed naturally and efficiently. Our AD pass supports a comprehensive set of features, including non-scalar types, pointers, and higher-order recursive functions.

We demonstrate through standard benchmarks that a suite of common optimizations effectively eliminates the overhead typically associated with functional AD approaches, producing differentiated code that performs on par with leading mutation and taping techniques. At the same time, MimIrADe's implementation is an order of magnitude less complex compared to its contenders.

CCS Concepts: • **Mathematics of computing → Automatic differentiation**.

Keywords: automatic differentiation, functional programming, intermediate representation

ACM Reference Format:
Marcel Ullrich, Sebastian Hack, and Roland Leißa. 2025. MimIrADe: Automatic Differentiation in MimIR. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3708493.3712685

1 Introduction

Automatic differentiation (AD) is a powerful tool for computing derivatives in numerical optimization, machine learning, and scientific computing. It enables efficient, accurate, and automatic gradient computation, eliminating the need for manual differentiation or numerical approximations.

Among various AD techniques, *reverse-mode AD* is especially notable for its efficiency in computing gradients of functions with *many inputs and only a few outputs*. It has become a fundamental building block for many modern machine-learning frameworks. In reverse-mode AD, the program first executes in a *forward pass*. Then, it is traversed backward from output to input, constructing a computation graph of derivatives (see Figure 1). The derivative of the output with respect to each input is calculated by recursively applying the chain rule, utilizing intermediate values stored in a *tape* during the forward pass. This tape records all intermediate computations and the execution paths taken, enabling efficient gradient computation with respect to any specified subset of inputs.

A common method to implement this taping approach is to overload arithmetic operators, recording intermediate values in the tape as they are calculated. During the backward pass, the tape is then accessed non-locally to efficiently compute gradients. In practice, modern AD frameworks like PyTorch [17] use dynamic computation graphs and sophisticated memory management techniques to minimize the memory overhead and optimize the performance of reverse-mode AD. Enzyme [16], on the other hand, is a state-of-the-art, fast, and highly efficient implementation of reverse mode AD that performs AD at compile time. However, Enzyme's performance comes at the cost of increased complexity, as it requires careful bookkeeping of intermediate values and other memory management techniques to reduce memory overhead. This makes Enzyme's implementation complex and hard to understand. In contrast, more theoretical work [e.g. 10, 21] typically presents higher-order functional AD with a set of modular, local rewrite rules.

The basic idea of these functional approaches is to augment a function to return a backpropagator [18] in addition

```
1  f(x,y):                8  x_d = 0, y_d = 0 // gradient initialization
2    tape['x'].           9  o_d = 1, z_d = 0
         push(x)          10 x_f = tape['x'].pop()
3    tape['y'].           11 z_f = tape['z'].pop()
         push(y)          12 x_d += z_f * o_d   // ∂o/∂x; line 7
4    z = x + y            13 z_d += x_f * o_d   // ∂o/∂z; line 7
5    tape['x'].           14 x_f = tape['x'].pop()
         push(x)          15 y_f = tape['y'].pop()
6    tape['z'].           16 x_d += 1 * z_d  // ∂z/∂x; line 4; x_d = z + x
         push(y)          17 y_d += 1 * z_d // ∂z/∂y; line 4; y_d = x
7    o = x * z            18 return (o, (x_d, y_d))
```

Figure 1. An imperative taping approach to compute the derivative for $x \cdot (x + y)$. The AD code is highlighted in blue.

```
λ (x,y).              λ ((x, x*), (y, y*)).
  let z = x+y in        let z, z* = (x+y, λs. x*(s) + y*(s)) in
  x * z                 (x * z, λs. x*(z*s) + z*(x*s))
```

Figure 2. Each computation of the original program (left) is augmented with a pullback marked with $\square^*$ (colored in blue, right). The differentiated function returns $x \cdot (x + y), \lambda s.s \cdot (2x + y, x)$.

to its original result (see Figure 2). The backpropagator, also called *pullback*, is a function that takes the derivative with respect to the function result, the *adjoint*, and returns the derivatives with respect to function inputs, the *tangent*. Expressing gradients as backpropagators enables the chain rule to be represented through function composition.

The transformation consists of a set of local rewrite rules applied in a bottom-up manner, converting the original function into its derivative. The functional, higher-order approach to AD with backpropagators offers a straightforward implementation and modularity, as the AD of a compound language element translates naturally into differentiating its components. However, the reliance on higher-order functions in gradient computation makes this approach more challenging for the compiler to produce efficient code.

Our approach, MimIrADe, combines a modular functional approach with an efficient implementation, using a higher-order intermediate representation (IR) and a set of standard optimizations to achieve state-of-the-art performance. At the heart of our approach is the use of the higher-order IR MimIR [14]. Because the IR is higher-order we leverage the simplicity of functional AD approaches and express AD by concise, local rewrites. The locality of the rewrites allows us to extend the implementation with high-level extensions like matrix operations in a modular way. Because the IR is graph-based, practically-relevant implementation aspects of functional approaches, most importantly handling the sharing of identical pullback invocations, basically come for free. Additionally, our AD handles a rich set of features including non-scalar types, pointers, and higher-order recursive functions.

1.1 Contributions

This paper makes the following contributions:

- MimIrADe differentiates a rich feature set including non-scalar types, pointers, conditionals, recursion, and high-order functions. Due to the modular nature of our approach, it is straightforward to extend MimIrADe to new high-level primitives such as matrix operations (Section 3).
- We demonstrate that MimIrADe achieves performance comparable to state-of-the-art contenders through evaluation on the ADBench benchmark suite (Section 4.1).
- At the same time, MimIrADe's implementation is an order of magnitude less complex than Enzyme's (Section 4.2).
- This is enabled by MimIR's optimization framework, which includes a partial evaluator. It succinctly removes the overhead of the local rewrites. We evaluate the optimizers of other compilers (Haskell, OCaml, Rust) with respect to their ability to optimize a simple program differentiated with the functional approach. We show that MimIrADe optimizes more aggressively and outperforms all of them (Section 4.3).

2 Background

This section provides a brief overview of the mathematical concepts of AD.

2.1 Reverse Mode Automatic Differentiation

Given a function f, the *gradient* ∇f is a function that evaluated at p produces the vector of partial derivatives of f at p. The partial derivative $\frac{\partial f}{\partial x}(p)$ of f at point p with respect to x is the rate of change of f at p in direction x. For a function with an n-dimensional domain and an m-dimensional co-domain the *Jacobian matrix* is defined as:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The goal of AD is to compute this matrix efficiently.

In *forward-mode* AD, the gradients are traced from the inputs to the outputs of the program through every computation, resulting in $\frac{\partial f}{\partial x_i}$ for some i hence computing a single column of the Jacobian. The gradient computation needs to be executed n times to compute the gradients with respect to every input x_i, and hence all columns of the Jacobian. Whereas the gradient is the derivative of the output for all inputs, the tangent is the derivative of all outputs for an input.

It is very common that $m \ll n$. Imagine a loss function when training a neural net. There, $m = 1$, i.e. the loss and n is large, i.e. the number of parameters in the neural net. Using forward AD one would still need n executions of the program to compute all n columns of the Jacobian.

This is why in practice *reverse-mode* AD is used more frequently. In *reverse-mode* AD, we start with the outputs and conceptually traverse the execution trace of the program in reverse (see Figure 1 and 2). Therefore, we get the gradient $\frac{\partial y_i}{\partial x}$ of one output y_i with respect to all inputs x in one reverse-mode AD computation. Thus, m evaluations suffice to compute the Jacobian matrix making this approach more efficient in the case of many inputs and few outputs:

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix} = \begin{bmatrix} \nabla^T y_1 \\ \vdots \\ \nabla^T y_m \end{bmatrix}$$

Pearlmutter and Siskind [18] proposed to use *backpropagators*—also called *pullbacks*—to implement reverse-mode AD. A pullback of a function f is a function f^* that internalizes the concept of reverse program traversal exhibited in reverse-mode AD. Conceptually, the pullback takes as argument the change in the output and returns the change in the inputs that lead to the change of the output.

Example 1. *Consider the pullback for $x \cdot y$:*

$$\begin{aligned} \texttt{mul}^* &\coloneqq \lambda s.\ (y \cdot s, x \cdot s) \\ \texttt{mul}^*(1) &= \begin{bmatrix} y & x \end{bmatrix}^T = \begin{bmatrix} \frac{\partial(x \cdot y)}{\partial x} & \frac{\partial(x \cdot y)}{\partial y} \end{bmatrix}^T = \nabla(x \cdot y) \end{aligned}$$

Note that the pullback reuses results from the forward pass of the computation.

As shown in the example, pullback functions are characterized by the equation:

$$f^*_{(x)}(s) = s \cdot \nabla f(x)$$

The formulation of gradients as pullbacks allows us to represent the chain rule as function composition of pullbacks.

Example 2. *Let $h = g \circ f$ with scalar functions f, g, and h. Then, h^* is defined as $f^* \circ g^*$:*

$$\begin{aligned} h^*(s) &\coloneqq f^*(g^*(s)) \\ &= g^*(s) \cdot \frac{\partial f(x)}{\partial x} \\ &= s \cdot \frac{\partial g(f(x))}{\partial f(x)} \cdot \frac{\partial f(x)}{\partial x} \\ &= s \cdot \frac{\partial g(f(x))}{\partial x} = s \cdot \frac{\partial h(x)}{\partial x} = s \cdot \nabla h(x) \end{aligned}$$

To compute gradients of programs in a modular fashion, we extend the notion of pullbacks to expressions. The pullback e^* of an expression e is the pullback of the associated function that computes the expression.

2.2 Automatic Differentiation as Local Substitution

Using pullbacks, reverse mode AD can be performed by local program rewrite. Local here means that a part of the program can be AD'd by AD'ing its constituents. For each function

$$f : A \to B$$

we generate its AD'd version

$$f^a : A^a \to B^a \times (B \to A)$$

that takes the input x and produces the output as well as the pullback for f at point x. We use the notation A^a for the augmented type. The augmented type represents additional information needed in the differentiated program. For instance, a higher-order function argument needs to provide its differentiated version. A^a is equal to A except for function types where we additionally return the pullback.

$$(A \to B)^a \coloneqq (A^a \to B^a \times (B \to A))$$

The differentiated function satisfies the equation

$$f^a\ x = (f(x),\ f^*_{(x)})$$

where $f^*_{(x)}$ is the pullback of f that produces the weighted derivatives in point x. We just write f^* and elide the application point if it is clear from the context.

Example 3.

$$\begin{aligned} f &\coloneqq \lambda(x,y).\ (x+y) \cdot y && \textit{of type} \quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ f^* &\coloneqq \lambda s.\ (s \cdot y,\ s \cdot (x + 2 \cdot y)) && \textit{of type} \quad \mathbb{R} \to \mathbb{R} \times \mathbb{R} \\ f^a &\coloneqq \lambda(x,y).\ (f\ (x,y),\ f^*) && \textit{of type} \quad \mathbb{R} \times \mathbb{R} \to \mathbb{R} \times (\mathbb{R} \to \mathbb{R} \times \mathbb{R}) \end{aligned}$$

Function f computes $(x+y) \cdot y$, f^a computes f and the pullback f^*. With the pullback, we compute the derivatives of f as $f^*\ 1$. The derivative with respect to x is y and the derivative with respect to y is $x + 2 \cdot y$.

We observe that the pullback is only closed in the context of the function. This means that f's parameters usually occur free in f^* as the pullback might use them directly or indirectly via intermediate results of f. Therefore, we compute the pullback, and thus the derivatives, and the function simultaneously in f^a. This allows for sharing between the function and the pullback to avoid redundant computations.

2.3 Algorithm

The classic pullback-based algorithm [18] implements reverse-mode AD as a local substitution of the individual language constructs. To compute f^a for a function

$$f \coloneqq \lambda a.\ b \quad \text{of type} \quad A \to B\,,$$

We recursively transform the body b of the function. For each composed expression, we construct the differentiated expression that uses the differentiated variants of the connectives. Our transformation keeps the structure of the program the same and augments every computation with the corresponding pullback.

Table 1. Pullbacks of arithmetic operations

$e = f$ args	f^*
`add` (x, y)	$\lambda s.\ (s, s)$
`sub` (x, y)	$\lambda s.\ (s, -s)$
`mul` (x, y)	$\lambda s.\ (s \cdot y, s \cdot x)$
`div` (a, b)	$\lambda s.\ (s/y, -s \cdot x / y \cdot y)$

Arithmetic Operations. These are replaced by their differentiated counterpart that contains the corresponding pullback. For example `mul` : $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ becomes:

$$\texttt{mul}^a \coloneqq \lambda(x, y).\ (\texttt{mul}\ (x, y),\ \underbrace{\lambda s.\ (\texttt{mul}\ (s, y),\ \texttt{mul}\ (s, x))}_{\texttt{mul}^*_{(x,y)}})$$

This represents the fact that

$$\partial(x \cdot y)/\partial x = y \qquad \text{and} \qquad \partial(x \cdot y)/\partial y = x\ .$$

With the differentiated multiplication `mul`^{a}, we compute the product and a pullback that returns a pair of partial derivatives. The derivative with respect to the left-hand side is y and the derivative with respect to the right-hand side is x:

$$\texttt{mul}^a(2, v) = (2v,\ \texttt{mul}^*) \qquad \texttt{mul}^*\ 1 = (v,\ 2)$$

Table ?? summarizes the pullbacks for arithmetic operations. As the shape of the differentiated operators is always similar, we will use a shorthand notation. The shape is in general:

$$f^a \coloneqq \lambda\text{args}.\ (f\ \text{args},\ f^*)$$

The pullback of a literal $c : S$ is given as

$$c^* \coloneqq \lambda s.\ \left(\vec{0} : A\right) \quad \text{of type} \quad S \to A\ .$$

3 MimIrADe

In the following section, we present the framework and the design and implementation decisions of our AD approach.

3.1 MimIR

We implement our AD in MimIR. This is an extension to the higher-order intermediate representation MimIR [13]. At its core, MimIR is a lambda calculus with a dependent, higher-order type system and contains lambdas, recursion, integers, and tuples. All other operations, like arithmetic, memory, pointers, floating point numbers, and loop constructs, are added through so-called *plugins*. A plugin contains typed declarations of the new operations, so-called axioms, as well as passes to operate on these axioms. As an example, Section 3.3.2 presents the axioms of a plugin that provides matrix computations. Passes include optimizations, lowering of axioms to the core calculus, and backends including a LLVM code generator.

MimIR represents a conditional by invoking a projection from a pair consisting of a "false"- and a "true"-continuation:

$$\textbf{if}(e)\ \textbf{then}\ \mathrm{T}\ \textbf{else}\ \mathrm{F} \qquad \Rightarrow \qquad (\mathrm{F}, \mathrm{T})\#e\ ()$$

The expression e selects the corresponding branch which is invoked with unit to continue execution. This way, our AD approach only needs rules for tuples and applications to also handle conditionals.

MimIR is well-suited for AD. The plugin system enables a simple definition of the differentiation operation. The minimal core calculus minimizes the burden to support all operations with the AD procedure. The dependent, higher-order type system allows for a succinct definition of gradients used in the AD. Furthermore, the optimization infrastructure of MimIR defines helpful standard optimizations including a partial evaluator that we utilize to generate efficient code for the gradients.

MimIR is a higher-order IR. This means that we simultaneously work on low-level code with a lot of control and optimizations for efficiency reasons as well as having features from functional programming available. The design of MimIR allows us to extend the language with the plugins by declaring high-level axioms. This also makes the implementation of the AD transformation easier.

3.2 Reverse Mode AD in MimIR

We encode the AD rules outlined in Section 2.2 straightforwardly in MimIR code using higher-order functions. We present the substitution rules for the core of MimIR and other common plugins in Section 3.3. For new plugins, the developer can provide differentiated versions of the axioms to extend the support of the transformation. Typically, the algorithm presented in Section 2.3 introduces a lot of overhead which MimIR reliably optimizes away by a series of optimizations that we discuss in more detail in Section 4.3.

Structure-Preserving Pullbacks. Along with normal pullbacks we also use a second kind called *structure-preserving pullbacks*. The idea behind them is to organize the pullbacks of individual elements of a data structure the same way the data structure is organized. *structure-preserving pullbacks* simplify the differentiation of operations that interact with compound types like `tuples` or `Ptrs`. For instance, for a projection into a tuple, we can access the pullback via a projection into the structure-preserving pullback. For an expression

$$e : C\ E \quad \text{using a compound type} \quad C : \mathsf{Type} \to \mathsf{Type}\ ,$$

the structure-preserving pullback is defined as

$$e^*_S : C\ (E \to A)\ .$$

Example 4. *The tuple* $t = (1, 4.3)$ *of type* $\underbrace{\mathbb{N} \times \mathbb{R}}_{T}$ *in a function*

$$f \quad \textit{of type} \quad \underbrace{\mathbb{R} \times \mathbb{R}}_{A} \to \mathbb{R} \textit{ has the pullback}$$

$$t^* \quad \textit{of type} \quad \underbrace{\mathbb{N} \times \mathbb{R}}_{T} \to \underbrace{\mathbb{R} \times \mathbb{R}}_{A}$$

and the structure-preserving pullback

$$t_S^* \quad \textit{of type} \quad (\underbrace{\mathbb{N}}_{\textit{fst } T} \to \underbrace{\mathbb{R}\times\mathbb{R}}_{A}) \times (\underbrace{\mathbb{R}}_{\textit{snd } T} \to \underbrace{\mathbb{R}\times\mathbb{R}}_{A}) \,.$$

A pointer p : $\texttt{Ptr}(\mathbb{R})$ *has the structure-preserving pullback*

$$p_S^* \quad \textit{of type} \quad \texttt{Ptr}(\mathbb{R} \to \mathbb{R}\times\mathbb{R}) \,.$$

The normal pullback t^* is a function taking the co-tangent with respect to the tuple and returning the tangent for the function. The structure-preserving pullback in contrast is a tuple of pullbacks for each of its elements.

Projections. The pullback of the projection $e = t\#i$ out of a tuple t with index i is the projection out of the structure-preserving pullback $e^* = t_S^*\#i$.

Tuples. For a tuple

$$t = (e_1, \ldots, e_n) \qquad \text{of type} \quad E_1 \times \cdots \times E_n$$

we need to define the pullback t^* as well as the structure-preserving pullback t_S^*. The pullback t^* of the tuple is the weighted sum of the pullbacks of the individual elements:

$$t^* = \lambda s_t.\ \sum e_i^*\ (s_t\#i) \quad \text{of type} \quad E_1 \times \cdots \times E_n \to A\,.$$

The structure-preserving pullback is the tuple of individual pullbacks

$$t_S^* = (e_1^*, \ldots, e_n^*) \quad \text{of type} \quad (E_1 \to A) \times \cdots \times (E_n \to A)\,.$$

Applications. We distinguish between two kinds of applications: For a return call `return` e, we also return the pullback `return` (e, e^*). For a function call $g\ e$, we call the differentiated function and compose the argument pullback with the function pullback:

```
1   (g e)^a :=
2       let (e, e*) = e^a in
3       let (y, g*) = g^a e in
4       (y, e* ∘ g*)
```

We first compute the argument and its pullback (line 2). Afterward, we compute the function call result and the pullback of the function (line 3). Then, we compute the pullback of the function application by composing the argument pullback with the function pullback (line 4). The main idea is that the pullback g^* computes the derivatives of y with respect to the function arguments e of g. The pullback e^* computes the derivatives of e with respect to the function argument a of f. By composing both of them, we compute the derivatives of y with respect to the function argument a of f. We take a closer look at the application rule in Section 3.1.

Example 5. *Consider the derivation process of* $f(a, b) = (a + b)\cdot b$ *in Figure 3a. The blue code segments in Figure 3b indicate that they were introduced by AD. This code mostly consists of boilerplate code for the tuple pullbacks and the pullback composition. Despite these changes, the control flow and overall shape of the program remain the same. Constant folding and inlining simplifies Figure 3b to Figure 3c as follows:*

```
fun f (a:ℝ, b:ℝ) → ℝ {
  let x = a+b in
  let y = x * b in
  return y
}
```

(a) original program

```
fun f^a (a:ℝ, b:ℝ) → ℝ {
  let a* = λs. (s,0) in
  let b* = λs. (0,s) in

  let (p,p*) = ((a,b), λ(s1,s2).
          a*(s1)+b*(s2)) in
  let (x, add*) = add^a p in
  let x* = p* ∘ add* in
  let (q,q*) = ((x,b), λ(s1,s2).
          x*(s1)+b*(s2)) in
  let (y,mul*) = mul^a q in
  let y* = q* ∘ mul* in
  return (y, y*)
}
```

(b) augmented version

```
fun f^a (a:ℝ, b:ℝ) → ℝ {
  let p = (a,b) in
  let x = add p in
  let q = (x,b) in
  let y = mul q in
  fun f* (s:ℝ) → ℝ×ℝ {
   let x* = (mul s b, mul s b) in
   let y* = x* + (0, mul s x) in
   // automatically folded to
   // (mul s b, mul s (add b x))
   y*
  }
  return (y, f*)
}
```

(c) optimized program

Figure 3. Differentiation of a function with two arguments

$$\begin{aligned}
y^* &= q^* \circ mul^* \\
&= q^* \circ (\lambda s.\ (s\cdot b, s\cdot x)) \\
&= \lambda s.\ x^*(s\cdot b) + b^*(s\cdot x) \\
&= \lambda s.\ (p^* \circ add^*)(s\cdot b) + b^*(s\cdot x) \\
&= \lambda s.\ p^*(s\cdot b, s\cdot b) + b^*(s\cdot x) \\
&= \lambda s.\ a^*(s\cdot b) + b^*(s\cdot b) + b^*(s\cdot x) \\
&= \lambda s.\ (s\cdot b, 0) + (0, s\cdot b) + (0, s\cdot x) \\
&= \lambda s.\ (s\cdot b, s\cdot (b + x)) \\
&= \lambda s.\ (s\cdot b, s\cdot (a + 2b))
\end{aligned}$$

This is the expected gradient $\frac{\partial f}{\partial a} = b$ *and* $\frac{\partial f}{\partial b} = a + 2b$ *for* $f(a, b) = (a + b)\cdot b = ab + b^2$*. The remaining gradient computation is as short as possible and corresponds to the code a human would write. Note that the last simplification is for readability;* x *would not be recomputed in the program.*

3.3 Extensions

Apart from arithmetic operations and connectives, we also want to support more advanced constructs beyond the most basic core of a programming language. MimIR in itself is a pure-type-system-style lambda calculus with dependent types. This core can be extended by declaring (usually compound and/or dependently typed) functions—called axioms since no implementation is given—and bundle transformations (such as constant folding) in *plugins*. In fact, numbers and arithmetic operations like addition and multiplication that we used in Section 2.3 are also implemented as a MimIR plugin.

To extend the AD transformation for a plugin we need AD versions of the plugin's axioms. These can be supplied with the plugin itself or added independently afterwards. In the next sections, we will take a look at some of the plugins that MimIrADe interacts with.

3.3.1 Memory Operations. MimIR's *memory* plugin introduces a memory object of type `Mem` that represents the memory state that must be used linearly (similar to the `IO` monad in Haskell). Furthermore, the plugin contains a type operator $\texttt{Ptr}(A)$ to construct a pointer to A and a set of operations to allocate and access memory:

$$\begin{aligned}
\texttt{alloc} &: \texttt{Mem} \to \texttt{Mem} \times \texttt{Ptr}(V) \\
\texttt{load} &: \texttt{Mem} \times \texttt{Ptr}(V) \to \texttt{Mem} \times V \\
\texttt{store} &: \texttt{Mem} \times \texttt{Ptr}(V) \times V \to \texttt{Mem} \\
\texttt{lea} &: \texttt{Ptr}(«i : n; T\#i») \times (i : \mathbb{N}) \to \texttt{Ptr}(T\#i)
\end{aligned}$$

Any *mutable* variable must be first `alloc`ated; this operation retrieves a pointer and the memory referenced by this pointer can be mutated via `load`s and `store`s. Note that memory operations take a `Mem` object as an argument and return a new `Mem` object in addition to their actual result. The `lea` operation[1] computes the address of the i^{th} element of an array or tuple. Hereby, $«n; T»$ is a *homogeneous* array with n elements of type T whereas $«i : n; T\#i»$ is a *heterogeneous* array (a.k.a. tuple type) with n elements where the element at index i has type $T\#i$.

Note that the structure-preserving pullback of a pointer is a pointer of a pullback. We use two strategies to differentiate pointers: *structure-preserving pullbacks* and *accumulation of gradients.*

Structure-Preserving Pullbacks. Our main strategy is to maintain the structure-preserving pullbacks:

- For an `alloc`, we additionally allocate the structure-preserving pullback pointer (see Figure 4a).
- If a pointer is `load`ed, the pullback from the structure-preserving pullback is loaded and associated with the pullback of the loaded value. But compared to the other axioms, we do not handle `load` by itself. Instead, we differentiate the application `load (mem, p)` (see Figure 4b). We first load the pullback of the pointer content from the structure-preserving pullback. This way, the loaded value is associated with the correct pullback. The invariant we use is that the structure-preserving pullback p_S^* always contains the pullback corresponding to the value contained in the pointer.
- Dually, if a value is `store`d in a pointer, the pullback of the value is stored in the structure-preserving pullback (see Figure 4c).
- For array accesses via `lea`, we index the structure-preserving pullback with the same index.

In this mode, the differentiation of memory operations composes with other operations. The only exception is that we use the application of the memory axioms as atomic units instead of the application and axiom differentiation individually, i.d. we differentiate load (mem, p) instead of load and mem separately. Dual to the forward pass, the backward pass uses memory side effects during the differentiation.

Accumulation of Gradients. Our second strategy is to accumulate gradients. This strategy is closely related to taping as explained in Figure 1. Similar implementations are employed by Moses and Churavy [16], who implement a taping approach, and Wang et al. [24], who use memory cells in which the gradient is computed using continuations. For read-only pointer arguments $p : \texttt{Ptr}(A_i)$, we do not maintain the structure-preserving pullback. Instead, we create a shadow pointer $p_G^* : \texttt{Ptr}(A_i)$ to accumulate the gradients. Its pullback adds the tangent to the associated shadow pointer (see Figure 5). Therefore, the gradient is accumulated in the backward pass of the uses of the pointer.

This optimization for pointer arguments helps to reduce normalization operations as the additions of tangents resulting from the pullbacks do not need to consider pointers. Instead, the accumulation takes care of the additions. Additionally, the access to pullbacks for arguments becomes simpler. With the gradient pullbacks, we can access the tangents directly instead of constructing structure-preserving pullbacks to wrap the identity for arguments.

The gradient pullback optimization is not strictly necessary as we could in principle obtain the same result from the main strategy. However, MimIR's partial evaluator is currently not powerful enough to transform the strategies into each other. For this reason, we optimize the argument case with the second strategy.

We apply the modified taping mode on the whole continuation to avoid issues with composition between the two modes.

3.3.2 Matrix Operations. The *matrix* plugin provides high-level operations on matrices. This plugin also allows for efficient code generation due to sophisticated optimizations on high-level operations.

Our transformation operates directly on the high-level matrix axioms rather than analyzing low-level loops and array accesses. This allows us to use more informed computations resulting in more efficient code. We look at the following common matrix operations:

$$\begin{aligned}
\texttt{unit} &: T \to \texttt{Mat}(T) \\
\texttt{sum} &: \texttt{Mat}(\mathbb{R}) \to \mathbb{R}
\end{aligned}$$

[1]The name is inspired by the x86 assembly instruction and its semantics is similar to `getelementptr` in LLVM but arguably more streamlined.

```
alloc^a := λ m_0.
  let (m_1, p*_S) = alloc m_0 in
  let (m_2, p) = alloc m_1 in
  ((m_2, p), λ (s_{m_0}, s_p). s_{m_0})
```
(a) alloc

```
(load (m_0, p))^a :=
  let (m_1, v*) = load (m_0, p*_S) in
  let (m_2, v) = load (m_1, p) in
  ((m_2, v), λ (s_m, s_v). v* s_v)
```
(b) load

```
store^a (m_0, p, v) :=
  let m_1 = store (m_0, p*_S, v*) in
  let m_2 = store (m_1, p, v) in
  (m_2, λ s_{m_0}. m*_0 s_{m_0})
```
(c) store

Figure 4. Differentiated axioms to handle memory operations

```
(load (m_0, p))^a :=
  let (m_1, v) = load (m_0, p) in
  ((m_1, v), λ (s_{m_0}, s_v). let (s_{m_1}, s_p) = load (s_{m_0}, p*_G) in
                             let s_p^a = s_p + s_v in
                             let s_{m_2} = store (s_{m_1}, p*_G, s_p^a) in p*_G)
```

Figure 5. Acc. of gradients w/ shadow pointer & array

$$\texttt{transpose} : \texttt{Mat}(T) \to \texttt{Mat}(T)$$
$$\texttt{product} : \texttt{Mat}(\mathbb{R}) \times \texttt{Mat}(\mathbb{R}) \to \texttt{Mat}(\mathbb{R})$$

For each of these operations, we provide a derived version that extends the differentiation transformation:

$$\begin{aligned} \texttt{unit}^a &\coloneqq \lambda e.(\text{unit } e, \lambda S.\ \text{sum} S) \\ \texttt{sum}^a &\coloneqq \lambda M.(\text{sum } M, \lambda s.\ \text{unit} s) \\ \texttt{transpose}^a &\coloneqq \lambda M.(M^T, \lambda S.S^T) \\ \texttt{product}^a &\coloneqq \lambda(M,N).(M \times N, \lambda S.(S \times N^T, M^T \times S)) \end{aligned}$$

We handle `unit` and `sum` as generalizations of addition and tuple construction. The pullback of the `transpose` is the `transpose` of the tangent whereas the pullback of the `product` is a `product` for each of its components. Note that MimIrADe just knows that it looks, for example, at a matrix product. This allows MimIrADe to simply use the optimal pullback and implement the product with highly-optimized computations.

For code generation, we translate matrices into arrays. The matrix operations are implemented on top of BLAS [4] or alternatively translated to straightforward loops nests.

3.3.3 Affine Operations. The *matrix* plugin lowers many operations to affine loops provided by the *affine* plugin:

$$\begin{aligned} \texttt{for} : \big(&\text{start} : \texttt{Int}, \text{end} : \texttt{Int}, \text{step} : \texttt{Int}, \text{acc}_{\text{init}} : T, \\ &\text{body} : (\texttt{Int} \times (\text{acc} : T)) \to T\big) \to T \end{aligned}$$

When applied with the loop bounds, the body, and the initial accumulator, the `for`-axiom returns the final accumulator. The body is applied $\lfloor \frac{\text{end}-\text{start}}{\text{step}} \rfloor$ times with the current index and the accumulator.

In general, we generate the differentiated code for the `for` axiom by first lowering it to loops with recursion. But for special cases, we generate more efficient gradients by performing a sophisticated partial evaluation manually via handcrafted optimization passes: We inspect the body to detect special cases like array manipulation that allow for more efficient code generation. If the body only reads and writes arrays and performs arithmetic operations, we simplify the caching behavior of the differentiated loop as follows:

For loops, we would generate closures around each iteration that capture the involved expressions. Instead, we create caches and only write the necessary values that we need to compute the tangents similar to Moses and Churavy [16]. The differentiated for-loop then consists of a forward pass for-loop with additional cache writes and a backward pass for-loop that computes the tangents. This allows us to avoid the memory allocation overhead of closures.

3.4 Implementation in MimIR

To better understand the details of the differentiation rules presented in Section 3.2, we take a closer look at the application rule. Let $f : A \to X$ be the function to be differentiated. If we stumble upon an application

$$y = g\ e \qquad \text{of type} \quad Y\,,$$

we compute

$$y, g^*_{(e)} = g^a\ e \qquad \text{of type} \quad Y \times (Y \to E)\,.$$

From the pullback $g^*_{(e)}$ with respect to the argument e of g and the pullback e^* with respect to the argument a of f, we compute the pullback of the application

$$y^* = e^* \circ g^*_{(e)} \quad \text{of type} \quad Y \to A$$

where

$$\begin{aligned} g &: E \to Y & e &: E \\ g^a &: E^a \to Y^a \times (Y \to E) & e^* &: E \to A\,. \end{aligned}$$

The same rule applies if g is the function f itself. In that case, we recurse.

4 Evaluation

We compare our implementation MimIrADe to the popular frameworks PyTorch [17] and Enzyme [16]. PyTorch builds dynamic computation graphs via operator overloading in Python. Enzyme differentiates LLVM code during compilation.

Although our approach is more similar to other high-level approaches known from functional programming languages from a theoretical point of view, we compare MimIrADe to the low-level implementations of state-of-the-art AD. We show that our modular functional approach competes with modern highly optimized implementations due to compiler optimizations that remove the overhead that originates from the functional approach.

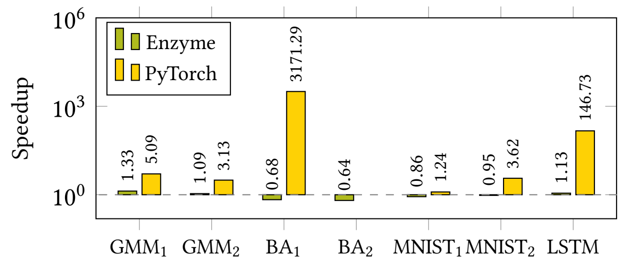

Figure 6. Speedup $t_{\text{framework}}/t_{\text{MimIR}}$ of MimIrADe vs. Enzyme and PyTorch

Setup. We evaluate the frameworks on Microsoft's ADBench suite [22]. We use the Gaussian mixture model (GMM), bounded analysis (BA), and a long short-term memory neural network (LSTM) from the ADBench suite. Additionally, we compare the approaches on a network for the MNIST classification task [8].

For a fair comparison to Enzyme, we do not take advantage of the high-level specialized extensions available in MimIR for matrix operations. Instead, we differentiate the code at the level of affine for-loops. This abstraction layer corresponds to the knowledge obtained by LLVM scalar evolution analyses in Enzyme.

4.1 Experimental Running Time

Figure 6 compares the runtime of the different implementations. For the MimIR implementations, we write the tests in the high-level language Impala that compiles to MimIR and used the MimIrADe AD compiler pass. Finally, MimIR emits an LLVM file that we feed to Clang to generate an executable. The Enzyme implementation[2] is written in C++ and uses the Enzyme LLVM pass. The PyTorch implementation[3] is written in Python and uses the PyTorch package.

Our evaluation shows that Enzyme and MimIrADe are comparable in performance. In some cases like bounded analysis, Enzyme has a better caching/recomputation balance resulting in lower runtime. In other cases like the GMM or LSTM benchmark, MimIrADe is faster due to more caching. The main performance difference is due to caching or recomputation of intermediate results. Enzyme manages to better detect induction variables and recompute them in the backward pass instead of storing them. But it also sometimes stores additional unnecessary intermediate results in the forward pass.

In our tests, PyTorch is slower than Enzyme and MimIrADe. This is partly due to the overhead of constructing the backward graph at runtime, the overhead of the Python interpreter, more memory usage, and missing optimizations like inlining. PyTorch is inefficient in the bounded analysis benchmark resulting in a significantly longer running time. This slowdown is caused by the deeply nested function calls and loops that contain conditionals.

TorchScript improves upon PyTorch by compiling parts of the code. The resulting speedup is especially visible for very large input sizes where TorchScript becomes one of the best implementations as shown in Figure 7. PyTorch 2.0 further refines the TorchScript approach of ahead-of-time compilation using TorchDynamo [2], TorchInductor, and AOTAutoGrad. For small inputs, the remaining overhead of Python, not compiled functions, and communication overhead still causes PyTorch 2.0 to be slower than Enzyme and MimIrADe. PyTorch 2.0 speeds up the computation for large input sizes by making use of available CPU cores. This can be seen in the slowdown when restricting PyTorch 2.0 to a single CPU. Enzyme and MimIrADe both do not make use of additional cores.

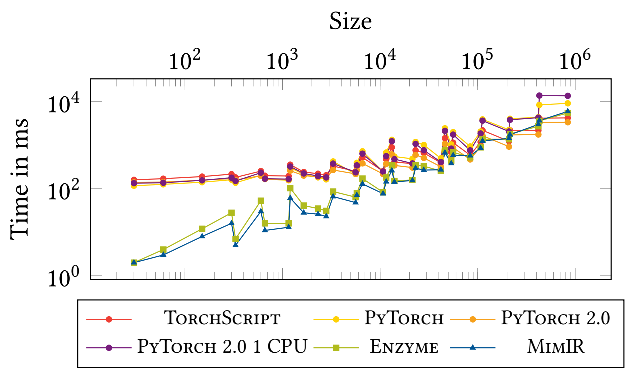

Figure 7. Runtime of the Gaussian Mixture Model (GMM) algorithm on different input sizes (lower is better). The diagram is a log-log plot.

Table 2. Lines of code (LoC) and cyclomatic complexity (CC) (lower is better). We compute the cyclomatic complexity measure per function. The total cyclomatic complexity refers to the sum over all files. ⬚† refers to the tool without plugins and extensions.

	MimIrADe	Enzyme	MimIrADe †	Enzyme †
LoC	4.7k	58.9k	1.3k	49.4k
Total CC	704	13793	141	11857
Average CC	1.6	10.9	1.8	12.8

4.2 Code Complexity

One major contribution of this paper is the simplicity of our approach. In order to verify this claim, we estimate the code complexity of MimIrADe and Enzyme using code complexity metrics. Table 2 summarizes the LoC. For Enzyme[4], we measured the source folder. To make the comparison fair, we also

[2]https://github.com/EnzymeAD/Enzyme/tree/main/enzyme/benchmarks/ReverseMode
[3]https://github.com/microsoft/ADBench/tree/master/src/python/modules/PyTorch
[4]https://github.com/EnzymeAD/Enzyme

measured the metrics for Enzyme † without the Clang plugin, the MLIR code, and without the different scalar evolution expander versions. With MimIrADe[5], we refer to the code of the AD plugin. MimIrADe † refers to the plugin without the special casing of pointer arguments. Enzyme has roughly 10× more code than MimIrADe. This does not necessarily give an indication of which code is simpler. But as a rule of thumb the larger a code base gets, the harder it becomes to debug and maintain. As a more profound code complexity metric, Table 2 also outlines the *cyclomatic complexity* [15][6] of both implementations. Enzyme's cyclomatic complexity is roughly an order of magnitude larger as MimIrADe's.

4.3 Effectiveness of MimIR's Optimizations

MimIrADe relies on MimIR's optimizer and, in particular, its partial evaluator to eliminate the boilerplate code introduced by the differentiation transformation (see Example 5). The optimizer is run before and after the differentiation transformation.

More specific, the program is optimized as follows: A set of peephole optimizations—called *normalizers*—are eagerly applied at every construction of a MimIR expression. These optimizations implement, for instance, constant folding and algebraic simplifications such as $x + 0 = 0$. Other optimizations like dead-code elimination and common subexpression elimination are performed automatically during the construction of a MimIR program. Finally, a set of standard optimizations is run. These include: • η-conversion • Tail call elimination • Scalarization of tuples • Copy propagation to specialize functions for statically known arguments • Partial evaluation: Function calls with statically known arguments cause beta-reduction to inline the call if possible. This optimization happens *independently* at each call-site. • β-reduction: On top of that β-reduction will always be performed if a function's sole use is a single call.

Together, these optimizations inline the pullbacks removing the boilerplate of closure allocation and function calls. Using copy propagation, partial evaluation, and β-reduction, functions are specialized as much as statically possible and are evaluated with the corresponding arguments. Statically known subexpressions will be simplified with the normalizers.

To empirically evaluate the influence of these optimizations and especially of MimIrADe, we compare the runtime of a deliberately simple program—the power function—before and after applying the optimizer.

Table 3 shows the runtime of the differentiated power function before and after applying the partial evaluation optimizations in MimIR, Haskell, OCaml, and Rust. For each test, we start with the MimIR code and transpile it to the other languages for comparison.

[5]https://github.com/NeuralCoder3/thorin2

[6]we used metrix++ for the measurement; see https://github.com/metrixplusplus/metrixplusplus

Table 3. Runtime in milliseconds of the differentiated power function before and after applying the partial evaluation (PE) optimizations in common languages (lower is better). FE stands for the fully evaluated derivative ($b \times a^{b-1}$). A gradient highlights slow (red) and fast (yellow) executions. MimIR* is the MimIR version compiled with Clang O3. Rust S refers to Rust with static dispatch.

	MimIR	MimIR*	Haskell	OCaml	Rust	Rust S
`PreOpt`	360	268	403	843		
`Optimized`	144	138	239	306		
`Manual`	145	148	153	710	785	179
`PE (Manual)`	155	148	199	80	117	100
`FE (Manual)`	17	1	22	24	1	1

- `PreOpt` corresponds to the original MimIR code right after the AD pass. This code includes all pullbacks as closures explicitly. We did not perform inlining, beta-reduction, or partial-evaluation at this stage.
- `Optimized` is the program that we obtain after running MimIR's optimizer on `PreOpt`. We inlined the pullbacks and simplified the code using partial evaluation.
- `Manual` is a simplified and specialized manual implementation of the differentiated power function in the respective language. This demonstrates what our approach would have generated in these languages. For this reason, we intentionally do not manually optimize these programs any further.
- `PartialEval (Manual)`, on the other hand, is a version where we *do* apply conceivable partial evaluation optimizations by hand to `Manual`. This code roughly corresponds to `Optimized` up to implementation details like continuation-passing style (CPS).
- `FullEval (Manual)` is a fully specialized manual implementation of the differentiated power function. This means, we directly compute $(a^b,\ (b \cdot a^{b-1},\ 0))$. Such optimized code is out of the reach of modern compilers due to the complex reasoning steps required.

The large difference between `ProOpt` and `Optimized` for MimIR highlights the importance of the optimizations. Without removing the boilerplate code, the code takes roughly twice as long.

We observe that MimIR performs all conceivable partial evaluation optimizations as witnessed by the coinciding runtimes of `Optimized`, `Manual`, and `PartialEval (Manual)`. In comparison, Haskell achieves a similar runtime as observed in `Manual`. However, other optimizations seem to be used as `PartialEval (Manual)` exhibits a worse performance. Additionally, Haskell is not able to achieve these runtimes in the original unoptimized MimIR code.

The OCaml compiler is unable to perform the optimizations resulting in slower code for most test cases. However, OCaml generates the fastest executable for the manually partially evaluated code.

Rust is similar to OCaml unable to perform the necessary partial evaluation optimizations. It also generates a very fast program for the manually partially evaluated code. In Rust, we observe a big difference between different encodings of the closures. Dynamic dispatch is used if we encode the closures as `Box<dyn Trait>` as shown in Rust `Release`. If we encode the closures where possible using generics and use static dispatch as shown in Rust `Static`, the performance is significantly higher. The latter encoding is closer to the code MimIR produces after closure conversion for code generation.

5 Related Work

Reverse mode AD requires the right-associative construction of the gradient, evaluating the chained derivatives from the back to the front. This evaluation can be achieved in multiple ways. One approach is constructing computation graphs during the execution to traverse the nodes explicitly [1, 17]. Another common approach is recording all necessary values and control-flow information using a tape [16]. This approach implements the traversal concretely without intermediate abstractions. These implementations commonly perform better than the more high-level methods but are more complicated due to handling non-local state.

High-level AD implementations compose derivatives directly, often representing them as closures [6, 9, 18, 24]. Vákár et al. [23] show that taping can be seen as optimizing this approach through partial evaluation and specialized data structures, mirroring our use of normalization. Similar connections are noted by Krawiec et al. [10] and their AD implementation.

Despite the choice of representing the derivatives, implementations can also vary in their details on how to implement the transformation itself. With the high-level, modular approach, operators can be replaced with their corresponding counterpart that also computes the derivative. This is commonly achieved using operator overloading [24] or compile-time passes involving rewrites [5, 9, 10]. In the implementation of Wang et al. [24], both forward and backward pass are interleaved using delimited continuations that compute the forward pass, call a continuation for the remainder program followed by the backward pass for the operator. Gradients are accumulated using memory cells. This implementation combines both passes at once with the help of memory cells for the gradients resulting in an implementation close to the lower-level taping approach but without an explicit tape.

Our approach separates forward and backward pass by using closures for the pullbacks that are combined to form the backward pass. This separation is made possible by handling application as a separate operator with its own derived form.

Shaikhha et al. [20] present an AD framework similar to AD in our high-level matrix plugin. They focus on the differentiation of array-processing primitives for which they develop sophisticated global optimizations such as loop transformations. Their forward pass AD combines advantages of reverse mode for many applications. In contrast, we focus on a simplified implementation keeping our code as modular and local as possible while achieving comparable performance.

Similar to Moses and Churavy [16] and Peng and Dubach [19], we operate on a low-level IR. Compared to LLVM [11], MimIR offers the advantage of having a much smaller core resulting in a simpler implementation. Additionally, MimIR makes it easier to define the transformation due to its functional nature. Similar to Moses and Churavy [16], we run optimizations before and after the AD transformation which results in asymptotically faster programs. As in the extensible compiler framework MLIR [12], MimIR allows for extensions making it possible to handle high-level structures like matrices directly in a more straightforward and yet more aggressive way. However, we found MimIR easier to use as MLIR due to full native support of higher-order functions wheras MLIR relies on other dialects [3] to fully represent them. On top of that, we argue that MimIR's "sea of nodes" simplifies the implementation and optimizations. The idea of a "sea of nodes" goes back to Click and Paleczny [7] while MimIR pioneered this concept for higher-order languages. The effectiveness of performing AD directly on high-level structures is demonstrated by Peng and Dubach [19].

6 Conclusion and Future Work

This paper presents a modular, high-level approach to AD through compiler transformation. This approach uses local substitutions to compute derivatives and eliminates boilerplate code through partial evaluation. Our experiments show that our method produces code that is as fast as state-of-the-art approaches while its implementation is significantly simpler and less complex than those of the best performing AD approaches. Additionally, we show that an extensible IR can provide high-level information (in our case through MimIR plugins) to choose optimal gradients for efficient code generation.

In the future, our AD approach can be extended to upcoming plugins of MimIR. We are also working on a MimIR plugin that adds metaprogramming capabilities. This would allow us to define the AD transformation completely inside of MimIR itself. Another area of work is to improve MimIR's optimizer in order to drop the special cases for argument pointers and for-loops (see Section 3.3.1) and let the partial evaluator do the heavy-lifting.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Jason Ansel. 2022. *TorchDynamo*. https://github.com/pytorch/torchdynamo

[3] Siddharth Bhat and Tobias Grosser. 2022. Lambda the Ultimate SSA: Optimizing Functional Programs in SSA. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 1–11. https://doi.org/10.1109/CGO53902.2022.9741279

[4] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[5] Timon Böhler, David Richter, and Mira Mezini. 2023. Using Rewrite Strategies for Efficient Functional Automatic Differentiation. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs*. 51–57. https://doi.org/10.1145/3605156.3606456

[6] Aloïs Brunel, Damiano Mazza, and Michele Pagani. 2019. Backpropagation in the simply typed lambda-calculus with linear negation. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–27. https://doi.org/10.1145/3371132

[7] Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, Michael D. Ernst (Ed.). ACM, 35–49. https://doi.org/10.1145/202529.202534

[8] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142. https://doi.org/10.1109/MSP.2012.2211477

[9] Conal Elliott. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29. https://doi.org/10.1145/3236765

[10] Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A Eisenberg, and Andrew W Fitzgibbon. 2022. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498710

[11] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[13] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsè ne Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: a partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 119:1–119:30. https://doi.org/10.1145/3276489

[14] Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2025. MimIR: An Extensible and Type-Safe Intermediate Representation for the DSL Age. 9, POPL (2025). https://doi.org/10.1145/3704840

[15] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[16] William Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. *Advances in neural information processing systems* 33 (2020), 12472–12485. https://doi.org/10.1109/SC41404.2022.00065

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d 'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[18] Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–36. https://doi.org/10.1145/1330017.1330018

[19] Mai Jacob Peng and Christophe Dubach. 2023. LAGrad: Statically Optimized Differentiable Programming in MLIR. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. 228–238. https://doi.org/10.1145/3578360.3580259

[20] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient differentiable programming in a functional array-processing language. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30. https://doi.org/10.1145/3341701

[21] Tom J Smeding and Matthijs IL Vákár. 2023. Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1573–1600. https://doi.org/10.1145/3571247

[22] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. 2018. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software* 33, 4-6 (2018), 889–906. https://doi.org/10.1080/10556788.2018.1435651

[23] Matthijs Vákár, Sam Staton, and Mathieu Huot. 2022. Higher order automatic differentiation of higher order functions. *Logical Methods in Computer Science* 18 (2022). https://doi.org/10.46298/lmcs-18(1:41)2022

[24] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–31. https://doi.org/10.1145/3341700

Received 2024-11-13; accepted 2024-12-21