LAGrad: Statically Optimized Differentiable Programming in MLIR

Mai Jacob Peng
McGill University
Canada
mai.peng@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
Canada
christophe.dubach@mcgill.ca

Abstract

Automatic differentiation (AD) is a central algorithm in deep learning and the emerging field of differentiable programming. However, the performance of AD remains a significant bottleneck in these fields. Training large models requires repeatedly evaluating gradients via AD potentially millions of times. Additionally, the most common form of AD incurs an asymptotically large memory cost relative to the original function being differentiated.

This paper introduces LAGrad, a reverse-mode, source-to-source AD system that leverages high-level information in MLIR to produce efficient differentiated code. LAGrad employs a collection of novel static optimizations that benefit from the semantics of high-level MLIR dialects to exploit the sparsity and structured control flow of generated code.

Using these, LAGrad is able to achieve speedups of up to 2.8× and use 35× less memory relative to state of the art AD systems on real-world machine learning and computer vision benchmarks.

CCS Concepts: • **Mathematics of computing** → **Automatic differentiation**; • **Software and its engineering** → *Source code generation.*

Keywords: automatic differentiation, MLIR, differentiable programming, static analysis, sparsity

ACM Reference Format:
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 (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3578360.3580259

1 Introduction

The widespread adoption of deep learning has led to a growing interest in using gradient-based optimization to learn parameterized *differentiable programs*. This new paradigm, known as *differentiable programming*, is a generalization of deep learning. It has already seen success in applications such as physics simulations [4], ray tracing [13], and many other fields [10, 12, 14].

Differentiable programming relies on the ability to automatically compute gradients of trainable parameters. The standard way to do this is via *Automatic Differentiation* (AD), which applies the chain rule to precisely compute derivatives, gradients, and Jacobians given only an objective function. AD avoids the disadvantages of finite differences and symbolic differentiation while relieving programmers from writing gradient code by hand [6, 21]. However, training via AD remains an expensive task which can involve millions of steps using gradient descent, requiring recomputing the gradient with respect to all model parameters at each step.

The current AD landscape consists of three orthogonal, contrasting axes, each of which are discussed in turn: 1) Operator-overloading vs source-to-source AD; 2) Forward mode vs reverse mode AD; and 3) Performing AD at different abstraction levels (high-level vs low-level).

Operator-overloading systems trace program execution at run time, transparently replacing operations with their differential versions. This sacrifices the potential for ahead-of-time optimizations of the gradient code. Source-to-source systems, on the other hands, analyze input programs to generate their differentiated versions at compile time. These systems have historically been less expressive than their operator-overloading counterparts, but recent work has shown renewed interest due to the potential for whole-program optimization of the generated code [9, 16, 20, 25], and this is the approach taken in this paper.

Forward-mode AD augments each step of the input program with a *dual* operations that compute derivative information in the same order as the original program. Unfortunately, forward-mode is prohibitively expensive for most ML applications as computing a gradient vector requires executing the forward sweep for each element of the gradient vector, which are typically numbered in the millions. For this reason, this work focuses on reverse-mode AD.

Reverse-mode AD involves running the original program to construct a *computation graph*, then propagating derivative information backwards through the graph. It is capable of computing an entire gradient vector in a single reverse sweep, but its inverted control flow means that some required intermediate values could potentially be overwritten. Reverse-mode AD thus requires a *gradient tape*: a data structure that records these intermediate values. This tape can become quite large and be detrimental to performance, even when so-called "tapeless" approaches are used as the intermediate values are still stored through mechanisms such as closures and delimited continuations [18, 22, 25].

Finally, AD can be applied on source languages of different abstraction levels. Popular ML frameworks such as PyTorch [17], JAX [5], and TensorFlow [1] perform AD on high-level multidimensional arrays. In contrast, Enzyme [16] differentiates at the low-level of LLVM IR, which means it can differentiate through programs written in a large number of source languages that target LLVM. However, the low-level nature of LLVM IR may hinder opportunity for novel, AD-specific optimizations.

This work introduces LAGrad, a source-to-source AD system that operates on high-level MLIR. LAGrad aims to maintain the generality of targeting a common compiler IR while preserving high-level information to facilitate the development of AD-specific optimizations. As we will see in this paper, LAGrad applies several optimizations such as tape size reduction and exploitation of sparsity using the information preserved in high-level MLIR.

The experimental results collected on CPU benchmarks demonstrate the benefit of these optimizations by achieving up to 2.8× speedup relative to Enzyme [16], the current state-of-the-art system, and up to 1400× speedup relative to PyTorch [17], a popular industry-standard ML library. LAGrad is also able to reduce memory consumption by up to 35× relative to Enzyme and 103× relative to PyTorch.

The main contributions of this paper are:

- LAGrad, an MLIR-based AD system that shows the advantage of using high-level information in source-to-source AD to generate efficient code.
- Three novel static optimizations (tape size reduction, active sparsity, and adjoint sparsity) that improve efficiency of reverse-mode AD.
- An evaluation of LAGrad against two state-of-the-art systems, PyTorch [17] and Enzyme [16] on a standard AD benchmark suite [21] that shows it outperforms the state of the art performance for both run time and memory consumption.

The rest of the paper is organized as follows. Section 2 gives preliminary overviews of both AD and relevant aspects of MLIR. Section 3 discusses unique characteristics of LAGrad's AD implementation due to the semantics of MLIR, in addition to a discussion of tape size reduction. Section 4 describes the static optimizations employed after the AD process is completed. Section 5 evaluates the efficacy of LAGrad's optimizations individually and against existing state-of-the-art methods. Section 6 discusses related work and the precise aspects of LAGrad that differentiate it from prior work, while Section 7 offers concluding remarks.

Primal	**Adjoint**
$z = wx + b$	$\overline{\sigma} = -(\overline{y})\dfrac{1}{\sigma^2}$
$\sigma = 1 + e^{-z}$	$\overline{z} = -(\overline{\sigma})e^{-z}$
$y = \dfrac{1}{\sigma}$	$\overline{w} = \overline{z}x$

Figure 1. An example of computing $\overline{w} = \frac{dy}{dw}$ using reverse-mode AD. The *primal* function is broken down into pieces, each of which are replaced with a pullback in reverse order to produce the differentiated *adjoint*. $\overline{y} = \frac{dy}{dy}$ is called the *seed* value, which is initialized to 1 when y is a scalar.

2 Background and Notation

2.1 Automatic Differentiation

Given a program that computes a function $y = f(x)$, the goal of reverse-mode AD is to compute the derivative $\frac{dy}{dx}$. The original program is called the *primal* while the program that computes the derivative is called the *adjoint*. The notation $\overline{z}$ is used to refer to $\frac{dy}{dz}$ for any input or intermediate value z.

AD accomplishes this using the chain rule of calculus:

$$\frac{dy}{dx} = \frac{dy}{dz_n}\frac{dz_n}{dz_{n-1}}\cdots\frac{dz_2}{dz_1}\frac{dz_1}{dx}$$

The primal consists of multiple simple operations (each producing a z_i intermediate value). The overall derivative is computed by propagating (via multiplication) information from the output back to the input in a *backward pass*.

Each differentiated version of a primal operation is called a *pullback* [9]. The pullback for the operation that produces z_i is a function that takes the propagated *gradient signal*, $\overline{z_i} = \frac{dy}{dz_n}\cdots\frac{dz_{i+1}}{dz_i}$, and the input(s) z_{i-1} to yield $\overline{z_{i-1}}$. An example is outlined in Figure 1. As can be seen, the adjoint expressions for $\overline{\sigma}$ and $\overline{z}$ contain dependencies on intermediate values computed in the primal (σ and z respectively). In general, the primal must execute to produce these values before the adjoint can be run. If these values are overwritten prior to their use in the adjoint, they must be explicitly saved to a data structure known as the *tape* [9]. The tape is a fundamental data structure of reverse-mode AD and incurs a memory overhead that does not exist in the original program.

When x and y are vectors, AD can be used to compute the Jacobian matrix. Reverse-mode AD computes the gradient of one element of y with respect to all elements of x (thus computing a row of the Jacobian) with a single backward pass. This property makes reverse-mode AD popular in ML applications, which typically have many trainable parameters and a scalar-valued objective function. Computing the full Jacobian of a function with m outputs requires m backward passes. Computing the ith row of the Jacobian requires initializing a seed vector using the ith column of the identity matrix (*i.e.,* 1 in the ith position and 0 elsewhere).

2.2 Tensors and the Linalg Dialect

Tensors in MLIR are an abstract representation of multidimensional arrays without a concrete underlying memory representation. Tensors are immutable, so operations on tensors that would update their values semantically return a copy to avoid mutating the underlying memory. Code that strictly uses tensors is thus free from side effects.

A tensor’s type contains an optional *encoding* — metadata used to describe tensor data. As we will see later, LAGrad uses this field to denote structured sparsity patterns.

Tensors are lowered to *memrefs*, multidimensional arrays that possess explicit underlying storage, in a process known as *bufferization*. Memrefs are mutable and must be explicitly allocated and deallocated. By default, bufferization will allocate new memrefs on every new `tensor` and `linalg` op to preserve that the immutability semantics of tensors.

The linalg dialect consists of operations that perform high-level linear algebra computation. These operations can all be expressed as the `linalg.generic` op, which is a general abstraction over parallel loop nests. It contains the following:

- One or more tensor inputs that map to outputs.
- An indexing map for each input and output, which describes how to index into each tensor from the iteration variables of each loop.
- An iterator type for each loop, which explicitly define which loops correspond to reductions of the outputs.
- A body, which is a basic block that defines what operations must be performed for each iteration of the innermost loop.

A `linalg.generic` op that takes N input tensors $T_1, \ldots, T_N$ with indexing maps $map_1, \ldots map_N$ and contains m iterators $d0, \ldots, dm$ to produce output tensor Out with indexing map map_O via body function f is expressed as follows:

```
for d0 from 0 to <inferred d0 bound>:
  ...
    for dm from 0 to <inferred dm bound>:
      $Out[map_O] = f(T_1[map_1], \ldots, T_N[map_N], Out[map_O])$
```

Observe that each indexing map is a function that maps $d0, \ldots, dm$ to a (possibly permuted) subset of its inputs. The loop bounds of each iterator are inferred by MLIR to iterate completely over the arguments.

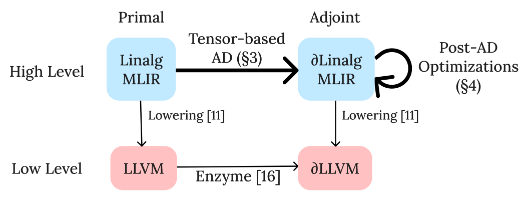

Figure 2. An overview of the approach taken in this work. The path of lowering before AD is the current state-of-the-art approach. The alternate path denoted by bolded arrows is taken by LAGrad, the contribution of this work.

For simplicity, this paper makes use of a Tensor Comprehensions (TC) [24] style notation to compactly show `linalg.generic` ops. Examples of full ops and their corresponding TC-style representations are shown later in Figure 3.

3 Auto. Differentiation on MLIR Tensors

This work proposes differentiating at the tensor MLIR [11] level, striking a middle ground between high-level operator overloading libraries and low level LLVM IR. An overview of this work is outlined in Figure 2. This remainder of this section outlines how differentiating at the MLIR level produces adjoint code that preserves the structure of the primal and is amenable to high-level optimizations.

MLIR expresses programs in Static Single Assignment (SSA) form [3], but differs from traditional SSA-form IRs by including the ability to express control flow through structured constructs (`scf.if`, `scf.for`, and `linalg.generic`) rather than basic blocks in a control flow graph. As we will see, the use of structured control flow enables AD-specific static optimizations that would be much harder to express on unstructured control flow graphs.

The complete set of operators supported by LAGrad consist of basic mathematical operators (`arith.addf`, `math.tanh`), if-statements (`scf.if`), and looping constructs (`scf.for`, `linalg.generic`). The process of differentiating scalar SSA programs in the absence of control flow is well documented, and we refer to [9] for a more in-depth discussion.

Basic mathematical MLIR operations like `arith.addf` can operate on scalars or tensors with identical shapes. These operations are *elementwise* where the same operation is performed for every element (or pair of elements) of its inputs. These can be differentiated identically to scalar arguments without special handling [9].

3.1 Differentiating Operators in MLIR

scf.if. To handle conditionals in the form of `scf.if` ops, LAGrad need only ensure that the condition value remains accessible so the adjoint can “replay” the branch that was taken (with the exception of `scf.if` ops inside loops, which is discussed later). LAGrad then collects free values that appear

```
%result = linalg.generic
  {  indexing_maps = [
       affine_map<(d0, d1, d2) -> (d0, d2)>,
       affine_map<(d0, d1, d2) -> (d2, d1)>,
       affine_map<(d0, d1, d2) -> (d0, d1)> ],
     iterator_types = [
       "parallel", "parallel", "reduction" ] }
   ins(%A, %B : tensor<?x?xf32>, tensor<?x?xf32>)
   outs(%C : tensor<?x?xf32>) {
     ^bb0(%arg0: f32, %arg1: f32, %arg2: f32):
       %0 = arith.mulf %arg0, %arg1 : f32
       %1 = arith.addf %arg2, %0 : f32
       linalg.yield %1 : f32 } -> tensor<?x?xf32>
```

$$C[i,j] \mathrel{+}= A[i,k] * B[k,j]$$

```
%pullback = linalg.generic
  { indexing_maps = [
       affine_map<(d0, d1, d2) -> (d0, d1)>,
       affine_map<(d0, d1, d2) -> (d2, d1)>,
       affine_map<(d0, d1, d2) -> (d0, d2)> ],
     iterator_types = [
       "parallel", "reduction", "parallel" ] }
   ins(%dC, %B : tensor<?x?xf32>, tensor<?x?xf32>)
   outs(%dA : tensor<?x?xf32>) {
     ^bb0(%arg0: f32, %arg1: f32, %arg2: f32):
       %0 = arith.mulf %arg0, %arg1 : f32
       %1 = arith.addf %arg2, %0 : f32
       linalg.yield %1 : f32 } -> tensor<?x?xf32>
```

$$dA[i,k] \mathrel{+}= dC[i,j] * B[k,j]$$

Figure 3. Left: A `linalg.generic` op corresponding to a matrix multiplication in MLIR. **Right:** Its corresponding pullback with respect to `%A`. The differences as a result of the AD process are highlighted.

in either branch of the `if` op and generates a corresponding adjoint `scf.if` op for each of them.

scf.for. The process to differentiate loops involves emitting an adjoint loop that iterates the same number of times as the primal, but in reverse. The loop body is then recursively differentiated with respect to both loop-carried iteration arguments and free variables.

`scf.for` loops in MLIR present the challenge of containing SSA values whose values will change over the course of the primal execution. The adjoint will often depend on these values from every iteration of the primal loop, meaning the values must be recorded to memory in a data structure known as the *tape* [8, 9]. The tape introduces a memory overhead that is avoidable in certain contexts, which is later discussed in Section 3.2.

linalg.generic. Differentiating a `linalg.generic` op will produce new `linalg.generic` ops. For simplicity, our discussion assumes each op produces one result. Multiple results involve performing the same procedure for each result.

Recall the following representation of a `linalg.generic` op:

```
for d0 from 0 to <inferred d0 bound>:
  ...
    for dm from 0 to <inferred dm bound>:
      $Out[map_O] = f(T_1[map_1],\ldots,T_N[map_N],Out[map_O])$
```

Suppose the pullback with respect to $T_i, 1 < i < N$ is desired. Its differential value $\overline{T_i}$ is assigned map_i, then f is differentiated to yield f' with respect to $T_i[map_i]$. This produces the following adjoint:

```
for d0 from 0 to <inferred d0 bound>:
 ...
  for dm from 0 to <inferred dm bound>:
   $\overline{T_i}[map_i]=f'(T_1[map_1],\ldots,T_N[map_N],\overline{Out}[map_O],\overline{T_i}[map_i])$
```

Iterator types of the adjoint are inferred by examining map_O. Input dimensions that appear in the map's results are marked `"parallel"`, while others are marked `"reduction"`.

A full MLIR code example of this process is found in Figure 3. The values `%dA` and `%A` use the same indexing map, as do `%dC` and `%C`. The iterator types are inferred based on map_A.

This differentiation procedure requires that the output argument of the `generic` op is not *effectively used*. The reason for this is further discussed in subsection 3.2 after covering the tape in more detail. If this condition is not met, a fallback is to immediately lower the `generic` op to an `scf.for` op and differentiate it as such.

3.2 Tape Size Reduction

A challenge of reverse mode AD (compared to forward mode) is the need to record primal values on the gradient tape. In some cases, these values can be recomputed instead of stored. LAGrad employs a novel static analysis that builds on prior work [8] to detect when it is beneficial to recompute.

A primal loop typically contains values that are overwritten during its execution. Given the immutable nature of SSA values in MLIR, these overwritten values must be explicitly represented in the `iter_args` of loops. For example:

```
func @f(%x: f32, %n: index) -> f32 {
  %r = scf.for %iv=0 to %n iter_args(%r_i = 0.0){
    %y = %iv * %iv : f32
    scf.yield %r_i + (%y * %x[%iv]) : f32 }
  return %r : f32 }
```

The adjoint loop may require these values (*e.g.*, `%y`) from all iterations, which requires their storage on the tape:

```
func @grad_f_v1(%x: f32, %n: index) -> f32 {
  %tape = memref.alloc(%n) : memref<?xf32>
  %r = scf.for %iv=0 to %n iter_args(%r_i=0.0){
    %y = %iv * %iv : f32
```

```
%slice = tensor.extract_slice %A[2] : tensor<4x5xf64> to tensor<5xf64>
%updated = linalg.generic ... outs(%slice) ...
%result = tensor.insert_slice %updated into %A[2] : tensor<5xf64> into tensor<4x5xf64>
```

Listing 1. A tensor program that exhibits the read/update/write pattern optimizable to an in-place update.

```
    memref.store %y, %tape[%iv]
    scf.yield %r_i + (%y * %x) : f32 }
  %dr= arith.constant 1.0 : f32
  %dx= scf.for %iv=%n-1 to -1 step -1 iter_args(%
    dx_i=0.0){
    %y = memref.load %tape[%iv] : f32
    scf.yield %dx_i + %dr * %y : f32 }
  return %dx : f32 }
```

%y can instead be cheaply recomputed to eliminate both the tape and the primal loop, leaving only the adjoint:

```
func @grad_f_v2(%x: f32, %n: index) -> f32 {
  %dr = arith.constant 1.0 : f32
  %dx = scf.for %iv=%n-1 to -1 step -1 iter_args(%
    dx_it=0.0) {
    %y = %iv * %iv : f32
    scf.yield %dx_it + %dr * %y : f32 }
  return %dx : f32 }
```

Observe that every loop contains some values that depend on previous iterations, and others that do not. For a loop ℓ, we define the following sets:

- $Vals(\ell)$: the set of values defined in the body of ℓ.
- $IterVals(\ell)$: the set of values that are carried through ℓ, or equivalently, the set of values that possibly have data dependencies on previous iterations of ℓ.

$IterVals(\ell)$ can be computed by traversing the iter_args of loops to find values that depend on the iter_args.

When loops are differentiated, the primal loop ℓ yields an adjoint loop $\partial\ell$. We define the set $AdjU(\ell)$ as the set of values defined in ℓ and used in $\partial\ell$. We say values in $AdjU(\ell)$ are *effectively used*. The computation of $AdjU(\ell)$ is prior work done by implementing a static analysis outlined in [8].

If $AdjU(\ell) \cap IterVals(\ell) = \emptyset$, then all required primal values in the adjoint loop can be recomputed with one iteration of the primal, signalling that recomputation is cheap. Otherwise, recomputation will require multiple primal iterations for each adjoint iteration, potentially incurring an asymptotically worse complexity than the original program.

If both recomputation is cheap and the result of the primal loop is not effectively used, LAGrad emits an adjoint that does not include the primal loop because it is unnecessary for the adjoint computation. Required primal values are recomputed within the body of the adjoint.

Other source-to-source AD systems are able to detect similar opportunities to avoid tape usage in simple examples via existing optimizations in LLVM [9, 16]. However, this new approach scales to arbitrarily complex code containing nested loops, conditionals, and linalg ops. We will see the effect this has on memory usage in the evaluation.

The potential presence of the tape when loop carried values are in $AdjU$ is why linalg ops cannot be differentiated when its output arguments are effectively used. The tape introduces a dependency on the primal iteration order, violating the parallel semantics of linalg iterators. This necessitates that such ops to be first lowered to sequential loops.

4 Post-AD Optimizations

Once adjoint code is generated, it can be optimized by the compiler prior to being lowered via bufferization to memrefs and then to LLVM. This section discusses optimizations performed immediately after AD is performed.

4.1 In-place Bufferization

Recall that MLIR's default bufferization will allocate new memory on each tensor op to preserve the immutability semantics of tensors. This process incurs a potentially large performance and memory overhead.

LAGrad contains several custom bufferization passes to avoid unneeded allocations and copies of memory due to MLIR's default bufferization. These passes focus on slicing operations on tensors, namely tensor.extract_slice and tensor.insert_slice. This work introduces *Insert-Extract (IE) Analysis* to find cases where a slice of a tensor is used as the output of a linalg.generic op before that slice is written to the same location in the original tensor. Consider Listing 1. Default bufferization creates new memory allocations for the extract, intermediate computation, and subsequent insertion:

```
// Allocate + copy for tensor.extract_slice
%slice = memref.alloc() : memref<5xf64>
%subview = memref.subview %A[2] : memref<4x5xf64>
    to memref<5xf64>
linalg.copy(%subview, %slice)
// Allocate + copy for linalg.generic
%updated = memref.alloc() : memref<5xf64>
linalg.copy(%slice, %updated)
linalg.generic ... outs(%updated) ...
// Allocate + 2 copies for tensor.insert_slice
%result = memref.alloc() : memref<4x5xf64>
linalg.copy(%A, %result)
%write_view = memref.subview %result[2] : memref<4
    x5xf64> to memref<5xf64>
linalg.copy(%updated, %write_view)
```

In contrast, IE bufferization results in the intermediate computation writing to memory directly in-place:

```
1  def f(x: float):
2    # Most elements of Y are constant, thus
       inactive
3    Y = np.zeros((2, 2))
4    Y[1, 0] = x
5    Z = Y ** 2
6    return Z.sum()
7
8  def grad_f(x: float, g: float):
9    Y = np.zeros((2, 2))
10   Y[1, 0] = x
11   dZ = np.broadcast_to(g, (2, 2))
12   dY = 2 * Y * dZ
13   # Inactive elements of dY are unused, meaning
       their computation could have been avoided.
14   dx = dY[1, 0]
15   return dx
```

Listing 2. Primal and adjoint functions containing sparsely active arrays, where most elements are constant w.r.t the input. Expressed in high-level Python for readability.

$$Y = \begin{bmatrix} 0 & 0 \\ x & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 0 & 0 \\ x^2 & 0 \end{bmatrix}$$

$$\overline{Z} = \begin{bmatrix} g & g \\ g & g \end{bmatrix} \qquad \overline{Y} = \begin{bmatrix} 0 & 0 \\ 2xg & 0 \end{bmatrix} \qquad \overline{x} = 2xg$$

Figure 4. Visualization of the sparsity of primal (top) and adjoint (bottom) values in Listing 2.

```
%slice = memref.subview %A[2] : memref<4x5xf64> to
     memref<5xf64>
linalg.generic ... outs(%slice) ...
```

To ensure the safety of this optimization, the destination of the insert must not have any uses that *postdominate* the insert. The accumulation of gradients in AD results in many such paired extract/insert ops, making this an important optimization in AD contexts.

4.2 Active Sparsity

Activity analysis is the process of determining which values can carry gradient signal from an input variable to an output variable. Determining activity is important as only active values require their gradients computed during AD [8]. However, most AD systems reason about activity at a coarse-grained level. When tensors are involved, this means that if a single element is active, the entire tensor is considered active. In practice, there are instances where only some elements of a tensor can propagate gradient information. This presents an optimization opportunity by only computing gradients for the active elements.

Consider the example in Listing 2. Lines 3 and 4 construct a 2 by 2 array where only the element at `[1, 0]` can carry a gradient signal. However, the entire array is involved in downstream computation and most AD systems compute gradients with respect to all elements of the array. This is shown in line 12, where the computation of $\overline{Y}$ involves 8 multiplications since both Y and $\overline{Z}$ are 2×2 arrays. Once computed, only one entry of $\overline{Y}$ is relevant to the gradient of the output, as shown on line 14.

```
// Ver. 1
func trmv_full(N, L, x, out):
  for i from 0 to N:
    for j from 0 to N:
      out[i] += L[i,j] * x[j]
// Ver. 2: Optimized compute, dense memory.
func trmv_triangular_computation(N, L, x, out):
  for i from 0 to N:
    for j from 0 to i:
      out[i] += L[i,j] * x[j]
// Ver. 3: Optimized compute, packed memory.
func trmv_packed_computation(N, Lpacked, x, out):
  for i from 0 to N:
    for j from i + 1 to N:
      Lidx = j - (i+1) + i * (2 * N - (i+1)) / 2
      out[j] += Lpacked[Lidx] * x[i]
```

Listing 3. Pseudo-code comparison of triangular matrix-vector multiplication. The compiler will automatically generate these examples from the same `linalg` op depending on the tensor encoding of L.

To address this, we define *sparsely active* tensors with the following criteria:

1. A significant portion of entries are zero.
2. Those same entries are constant with respect to the input, and as such their gradient values will be unused.

LAGrad optimizes these sparsely active tensors when the patterns of active elements follow a statically known shape, such as being in the lower or upper triangular portion of the tensor. Note that the values in the example in Listing 2 satisfy this property in the lower triangular case.

To perform these optimizations, the IR must "merely" be annotated such that the tensor encoding contains the sparsity pattern of the operand. The compiler then transforms `linalg` ops that contain a sparse operand into loops that iterate over the nonzero elements of their operands by modifying the resulting loop bounds. This transformation optimizes computation while leaving zero values materialized in memory.

Building on this, LAGrad contains an additional optimization that automatically convert triangular tensors into *packed* representation, such that only nonzero values are stored in memory [2]. This process automatically generates code to map iteration variables to the indexing scheme of packed triangular storage. These methods are shown in Listing 3. The packing of lower triangular tensors potentially improves cache locality of computation operating on these tensors.

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

$$C[j,i] \mathrel{+}= A[i,k] * B[k,j]$$

$$M = \begin{bmatrix}\bullet&\bullet&\bullet\\ \bullet&\bullet&\bullet\\ \bullet&\bullet&\bullet\end{bmatrix}, \quad N = \begin{bmatrix}&\bullet&\\ &\bullet&\\ &\bullet&\end{bmatrix} \implies O = \begin{bmatrix}\bullet&\bullet&\bullet\\ \bullet&\bullet&\bullet\\ \bullet&\bullet&\bullet\end{bmatrix}$$

$$O[i,j] \mathrel{+}= M[i,k] * N[j,k]$$

Figure 5. Examples of propagating per-dimension sparsity patterns through `linalg` ops. • represents a nonzero entry. The sparsity of C and O are automatically inferred through the sparsity of the inputs and the indexing maps of each `linalg` op. Opportunities for sparse code generation are present when computing O despite its lack of sparsity.

These optimizations are not intrinsically linked to AD, but will be applied to both the primal and adjoint versions of ops containing these sparse tensors. This is due to how the gradient of every value in LAGrad has the same type as the primal value. Thus, primal values marked lower triangular will also have their gradients automatically annotated lower triangular, resulting in both benefiting from the optimization.

4.3 Adjoint Sparsity

In addition to active sparsity, sparsity arises in AD when computing full Jacobian matrices. Recall that the process of computing a Jacobian matrix involves repeated backward passes using columns of the identity matrix as seed vectors.

These seed vectors, when used as an argument in `linalg` ops, result in many highly sparse intermediate values. Crucially, these sparse values follow predictable patterns such as having a single element, row, or column be nonzero. The notion of sparsity is tied to each dimension, where a sparse dimension can contain at most one location where nonzero values appear. For example, in the two-dimensional case, a tensor with two sparse dimensions will only contain one nonzero element. A tensor with [sparse, dense] dimensions will contain a single nonzero row, while a tensor with [dense, sparse] dimensions will contain a single nonzero column. We refer to these tensors as having *one-hot dimensions.* This notion can be extended to *few-hot dimensions* when there is a small number of valid indices per dimension that can contain nonzero elements.

Sparse Propagation. A key property of one-hot and few-hot tensors is that their use in `linalg` ops results in *propagation* of sparsity. As sparsity is tied to dimensions, the sparsity of `linalg` results is statically determined as follows:

- For every `linalg` op in a program with input tensors $InTensors$ and output tensors $OutTensors$, let $Dims(t)$ be the set of loop dimensions for $t \in InTensors \cup OutTensors$.
- Let $SparseDims(t)$ be the set of loop dimensions that iterate over a sparse dimension of t.

$\forall o \in OutTensors$, sparse dimensions are computed as:

$$SparseDims(o) = \bigcup_{t \in InTensors} SparseDims(t) \cap Dims(o)$$

For example, consider the op in the first example of Figure 5, where the second argument B is sparse along both dimensions. This op has these indexing maps for B and C:

$$map_B = (d0, d1, d2) \rightarrow (d2, d1)$$
$$map_C = (d0, d1, d2) \rightarrow (d1, d0)$$

This results in the following:

$$Dims(B) = SparseDims(B) = \{d2, d1\}$$
$$Dims(C) = \{d1, d0\}$$
$$\implies SparseDims(B) \cap Dims(C) = \{d1\}$$

The final result is that C has dimensions [sparse, dense], meaning it contains a nonzero row. This procedure is run top-down from every function.

Code generation. After sparse propagation analysis, LAGrad lowers `linalg.generic` ops with sparse inputs to loops that skip over zero values. To this end, LAGrad stores the indices of nonzero positions for sparse dimensions of tensors within the compiler. The code generation pipeline uses these internal data structures to index into sparse tensors.

Indices are propagated in the same way as sparsity for dimensions. In the previous example, the sparse indices for the first dimension of C are the same as the indices for the second dimension of B.

5 Evaluation

The performance of LAGrad is evaluated in two stages. First, individual optimizations are selectively disabled to examine the effect of each optimization's individual contribution. Then, the fully optimized version is evaluated against both Enzyme [16] and PyTorch [17].

LAGrad and Enzyme both perform source-to-source AD in a compiler infrastructure, but they differ on the level of abstraction of their input and output IR. PyTorch performs AD on a high level of abstraction, but its AD implementation is operator-overloading based.

5.1 Experimental Methodology

The performance of all AD systems are evaluated using ADBench [21], a standard machine learning benchmark suite. ADBench consists of a gaussian mixture model (GMM), bundle adjustment (BA), a hand tracking model (Hand), and a long short term memory network (LSTM).

Datasets for each ADBench benchmark are chosen in figures to demonstrate the performance of the smallest problem

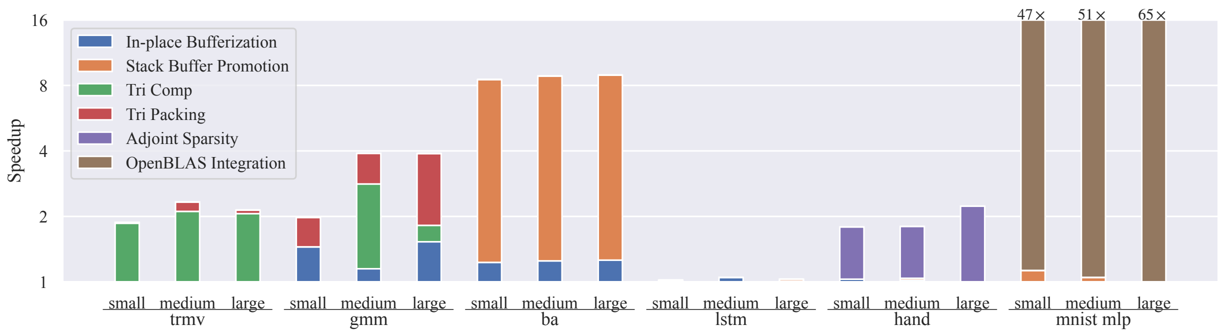

Figure 6. Performance impacts of individual optimizations in the LAGrad pipeline. The baseline used is LAGrad run through -O3 without any of the custom optimizations implemented in this work.

size that both Enzyme and LAGrad require more than 5 milliseconds to finish (labelled *small* in the figures), the largest problem size that both tools can finish within 40 minutes (labelled *large*), and the median problem size between the two (labelled *medium*). Measurements for smaller datasets introduce higher variance in the results, while runs longer than 40 minutes are considered timeouts.

A triangular matrix vector multiplication (TRMV) is used to measure the isolated effect of active sparsity, and a two-layer multi-layer perceptron (MLP) is used to evaluate LAGrad on a classical neural network application. The evaluated sizes are [1024, 2048, 4096] for the TRMV benchmark and hidden size [256, 512, 1024] for the MLP benchmark.

All experiments are run on a 2015 MacBook Pro with a 2.2 GHz Quad-Core Intel Core i7 processor and 16 GB of RAM. The operating system is macOS Catalina version 10.15.5.

Run time evaluations are measured by taking the median runtime of running each benchmark 5 times with 1 warmup run. Memory consumption is measured by taking the peak resident set size during the execution of each benchmark. Relative memory reduction is reported, where a value of 2 means LAGrad used 2× less memory than the compared tool.

Baseline LAGrad. Each benchmark from the ADBench suite is translated by hand to high level MLIR in the `linalg`, `tensor`, and `scf` dialects. They are then run through LAGrad to differentiate. The generated adjoints are first bufferized to linalg-on-memrefs, then lowered to loops in the `scf` dialect before being lowered to the LLVM dialect. Finally, the programs are translated to LLVM IR, then compiled to object files with `clang` using the -O3 optimization level.

5.2 Optimizations

Optimizations presented in this work are evaluated by enabling their respective flags to augment the baseline. The program is evaluated after enabling each optimization in the given order: in-place bufferization, stack buffer promotion, active sparsity, adjoint sparsity, and library call integration. Results of these optimizations are summarized in Figure 6.

Active Sparsity is evaluated in two stages: 1) Triangular computation is optimized while sparse tensors are left fully materialized (*Tri Comp*) 2) Sparse tensors are packed to store only nonzero entries (*Tri Packing*). The benchmarks that present opportunities for active sparsity are the TRMV and GMM benchmarks. The GMM benchmark contains a triangular matrix vector multiplication in addition to computation of matrix norms of triangular tensors. The TRMV benchmark primarily benefits from optimizing computation with modest cache locality gains from packing, while the cache benefit of packing is more strongly felt in the GMM benchmark due to the matrix norm computations.

Adjoint Sparsity benefits the computation of full Jacobian matrices. Hand tracking is the one evaluated benchmark that performs this, which results in a number of dimension-level sparse values. The resulting speedup of sparse code generation is increased for larger datasets which have Jacobians with a greater number of rows.

Stack Buffer Promotion is a built-in MLIR pass that promotes memref allocations statically known to be below a set size from the heap to the stack. This can improve performance when all buffers required to compute an adjoint are small, such as in the BA benchmark, which consists entirely of computation on tensors with fewer than 12 elements. Applying stack buffer promotion to BA results in the adjoint program allocating memory entirely on the stack, leading to the speedup observed in Figure 6.

Library Call Integration. The high-level nature of `linalg` ops in MLIR makes it straightforward to target optimized linear algebra libraries. LAGrad has basic support for replacing named ops in the `linalg` dialect (`linalg.matmul`, `linalg.matvec`) and their pullbacks with calls to OpenBLAS [26] routines.

5.3 Comparison with State of the Art

After evaluating individual optimizations, we now turn our attention to comparing the most optimized LAGrad variant with Enzyme [16]. Results are summarized in Figure 7.

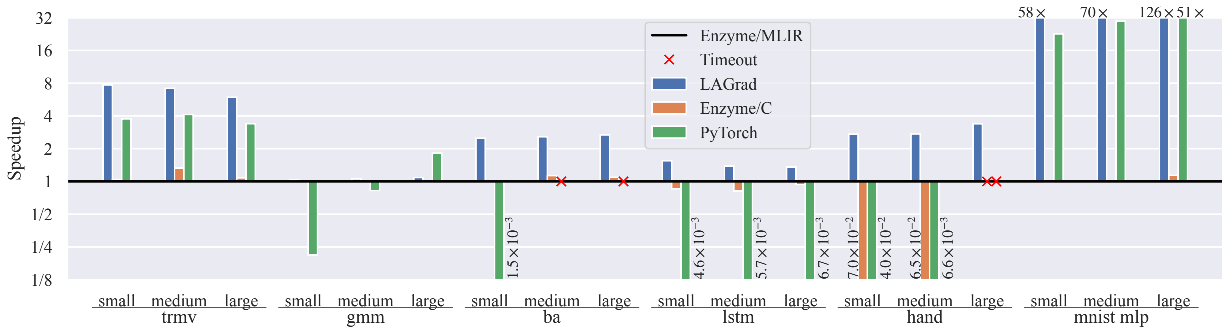

Figure 7. Speedup of the fully optimized LAGrad variant vs Enzyme and PyTorch. The baseline is Enzyme performing AD on the translated benchmarks in high-level MLIR, while Enzyme performing AD on the original C programs from ADBench is also included. A red × indicates that a benchmark did not complete within the allotted time limit.

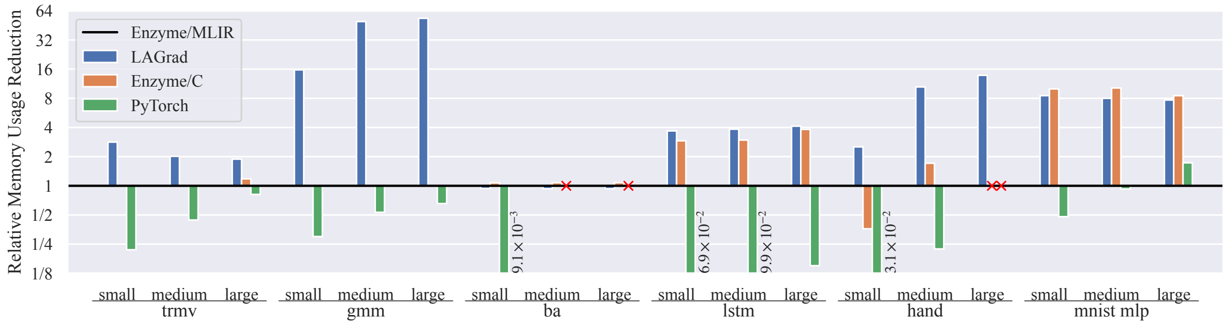

Figure 8. Relative peak memory use reduction of LAGrad vs Enzyme and PyTorch (higher is better).

Table 1. Geometric mean speedups and relative memory reduction of each benchmark across all evaluated datasets.

Benchmark	Speedup w.r.t. Enzyme (MLIR)	Memory reduction w.r.t. Enzyme (MLIR)	Speedup w.r.t. PyTorch	Memory reduction w.r.t. PyTorch
TRMV	6.9	2.2	1.8	5.1
GMM	1.1	35.0	6.4	74.0
BA	2.1	0.9	1419.1	103.1
Hand	2.8	7.8	168.7	61.0
LSTM	1.5	3.9	268.6	38.6
MLP	79.6	8.0	2.3	8.8
Geomean	**4.2**	**5.2**	**34.6**	**30.5**

Enzyme is evaluated with two different pipelines for completeness. The first pipeline begins from the MLIR translation of each benchmark, while the second begins from the C implementations provided in ADBench. Both are lowered to LLVM IR before being run through Enzyme's optimization pipeline outlined in [16]. The purpose of including both pipelines is to compare Enzyme and LAGrad from the same starting program, while also comparing against the baseline C implementations evaluated in [16].

TRMV demonstrates one of the benefits of performing AD on `linalg` ops. Enzyme differentiates TRMV as a nested loop without the context of it being a linear algebra operation. This results in it storing a value to the tape for every loop iteration, using $O(n^2)$ memory. In contrast, LAGrad produces a `linalg` op as its pullback with no memory overhead. LAGrad is then able to remove the primal op as it is unneeded, an optimization that both PyTorch and Enzyme are unable to perform in this case. PyTorch cannot remove the primal op because its run-time AD requires execution of the complete primal to track which ops to differentiate. LAGrad outperforms PyTorch via active sparsity, in spite of PyTorch's usage of high performance libraries.

Gaussian Mixture Models (GMM). LAGrad displays comparable performance with Enzyme on Gaussian Mixture Models. Notably, the GMM benchmark contains operations on actively sparse lower triangular tensors. Both Enzyme and LAGrad use packed representations, but LAGrad automatically generates packed code from annotated `linalg` ops (Ver. 3 of Listing 3) while in Enzyme, the primal must be manually coded with these index computations for full performance.

Bundle Adjustment (BA). The speedup over Enzyme with bundle adjustment is due to stack buffer promotion. Enzyme crucially cannot benefit from the same optimization due to the unstructured nature of the control flow graphs of LLVM IR. Buffers allocated on the stack in the primal are often moved to heap allocations by Enzyme to ensure that they are accessible in the adjoint. However, LAGrad preserves the structured control flow of the primal when generating the adjoint, making this optimization safe to perform.

Long Short Term Memory (LSTM). The performance difference of LAGrad over Enzyme is primarily due to the difference in memory usage. The MLIR variant of Enzyme is penalized by naive bufferization, as Enzyme must produce gradients of every intermediate buffer. LAGrad does not have this issue by virtue of operating at the tensor level, where memory is abstracted.

Hand Tracking (Hand). Hand tracking involves a full Jacobian computation. It thus benefits from the propagation and code generation of adjoint sparsity. The benefit is more pronounced as the size of the Jacobian increases, and would be much more challenging to implement in Enzyme due to needing to recover the high level information that is directly included in `linalg` ops in MLIR.

Multi-Layer Perceptron (MLP). The performance of the MLP benchmark is almost entirely dominated by dense linear algebra kernels, something that Enzyme is currently unable to efficiently differentiate. LAGrad and PyTorch both leverage libraries to outperform Enzyme. The speedup LAGrad observes over PyTorch is due to the performance of OpenBLAS, which LAGrad uses, over the PyTorch CPU backend.

6 Related Work

The most commonly used methods of automatic differentiation in ML are based on operator overloading (OO). These include PyTorch [17], TensorFlow eager [1], Autograd [15], and JAX [5]. As OO-based methods perform AD by tracing program execution at runtime, they give up the potential for whole program optimization that is possible in source-to-source methods. These methods will implicitly unroll loops, losing any structured control flow present in the primal. This hinders the opportunity to perform optimizations such as tape size reduction in these systems.

Zygote [9] performs source transformation on an SSA-form IR in the Julia compiler. Unlike MLIR, this IR is not a compilation target supported by multiple frontends. In general, the compiler infrastructure of MLIR makes it straightforward to implement new frontends to LAGrad.

Tapenade [7] supports both forward and reverse mode AD on Fortran and C code. Its approach to differentiating structured control flow greatly influenced LAGrad. It reasons about arrays merely as pointers, making it challenging to support array-level optimizations to exploit sparsity.

Tangent [23] is similar to LAGrad in performing source-to-source AD on high level code. It boasts improved debugging by generating readable Python code, but incurs interpreter overhead for both primal and adjoint code. This presents challenges in performance critical ML workloads.

Fsmooth [20] differentiates a functional array language using compiler rewrite rules to produce optimal derivative expressions. The level of abstraction of functional primitives carries less static information than MLIR. It also has the limitations of forward mode AD when computing gradient vectors of many parameters.

To the best of the authors' knowledge, no other AD system operates on immutable arrays with the semantics of tensor MLIR. As we have shown, this reduces the complexity of performing AD-specific static analyses and optimizations.

7 Conclusion

This work demonstrates the benefit of performing source-to-source AD on a high level language. Some of the optimizations presented would be infeasible to implement in operator-overloading based systems, while others would be challenging to rediscover in low level languages. LAGrad is able to discover opportunities to elide the gradient tape, long considered a fundamental challenge of reverse-mode AD.

By targeting the compiler *lingua franca* of MLIR, LAGrad can benefit from the ecosystem that MLIR provides. The MLIR infrastructure makes it straightforward to define new frontends that target the subset of MLIR that LAGrad can differentiate. Motivations of MLIR include improving reusability of high-level compiler optimizations and targeting heterogeneous hardware accelerators. As MLIR matures towards these goals, LAGrad may further benefit in tandem.

Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

A Data Availability Statement

The software and data that support the findings of this paper are openly available within Figshare [19] and released under a CC-BY 4.0 license.

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users’ Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[4] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J. Zico Kolter. 2018. End-to-End Differentiable Physics for Learning and Control. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/842424a1d0595b76ec4fa03c46e8d755-Paper.pdf

[5] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[6] Andreas Griewank et al. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.

[7] Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software* 39, 3 (2013). https://doi.org/10.1145/2450153.2450158

[8] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. 2005. “To be recorded” analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems* 21, 8 (2005), 1401–1417. https://doi.org/10.1016/j.future.2004.11.009

[9] Michael Innes. 2018. Don’t Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR* abs/1810.07951 (2018). arXiv:1810.07951 http://arxiv.org/abs/1810.07951

[10] Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B Shah, and Will Tebbutt. 2019. A Differentiable Programming System to Bridge Machine Learning and Scientific Computing. https://doi.org/10.48550/ARXIV.1907.07587

[11] 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)*. IEEE, 2–14.

[12] Li Li, Stephan Hoyer, Ryan Pederson, Ruoxi Sun, Ekin D. Cubuk, Patrick Riley, and Kieron Burke. 2021. Kohn-Sham Equations as Regularizer: Building Prior Knowledge into Machine-Learned Physics. *Phys. Rev. Lett.* 126 (Jan 2021), 036401. Issue 3. https://doi.org/10.1103/PhysRevLett.126.036401

[13] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. 2018. Differentiable Monte Carlo Ray Tracing through Edge Sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)* 37, 6 (2018), 222:1–222:11.

[14] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable Programming for Image Processing and Deep Learning in Halide. *ACM Trans. Graph.* 37, 4, Article 139 (jul 2018), 13 pages. https://doi.org/10.1145/3197517.3201383

[15] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2015. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML workshop*, Vol. 238.

[16] William Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12472–12485. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

[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*. 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 Trans. Program. Lang. Syst.* 30, 2, Article 7 (mar 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

[19] Mai Jacob Peng. 2023. lagrad-artifact.tar.gz. (1 2023). https://doi.org/10.6084/m9.figshare.21964787.v1

[20] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (jul 2019), 30 pages. https://doi.org/10.1145/3341701

[21] 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 (02 2018), 1–14. https://doi.org/10.1080/10556788.2018.1435651

[22] Bart van Merrienboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. 2018. Automatic differentiation in ML: Where we are and where we should be going. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/770f8e448d07586afbf77bb59f698587-Paper.pdf

[23] Bart van Merrienboer, Dan Moldovan, and Alexander Wiltschko. 2018. Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/748d6b6ed8e13f857ceaa6cfbdca14b8-Paper.pdf

[24] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[25] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *Proc. ACM Program. Lang.* 3, ICFP, Article 96 (jul 2019), 31 pages. https://doi.org/10.1145/3341700

[26] Zhang Xianyi, Wang Qian, and Zhang Yunquan. 2012. Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 684–691. https://doi.org/10.1109/ICPADS.2012.97

Received 2022-11-10; accepted 2022-12-19