High-Level Static Optimizations for Efficient Differentiable Programming in MLIR

Mai Jacob Peng

School of Computer Science
McGill University
Montréal, Québec, Canada
May, 2023

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Automatic differentiation (AD) is ubiquitous in the training of deep neural networks and other machine learning tasks. The emerging field of *differentiable programming* has recently found success in generalizing deep learning by applying gradient-based optimization via AD to increasingly sophisticated applications in a diverse array of fields [1–5].

However, the most popular tools for AD are hyper-specialized to deep learning workloads. The performance of both the AD process itself and the resulting differentiated code suffer as applications veer further away from deep neural networks. The most popular AD tools also perform differentiation at runtime, which incurs runtime overhead with each of the potential millions of gradient descent steps. Some of these issues can be mitigated through performing ahead-of-time AD in a compiler. However, existing compiler-based methods predominantly operate on low-level compiler intermediate representations (IRs) that lose context and information after being lowered from the original program. Additionally, the most common form of AD incurs an asymptotically large memory cost relative to the original program, regardless of if the AD procedure is done at compile time or run time.

To address these challenges, this thesis introduces LAGrad, a reverse-mode, compile time 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. LAGrad is the first tool to the authors' knowledge to exploit the structure and sparsity inherent in AD through static optimizations in a compiler.

Abrégé

La différenciation automatique (DA) est omniprésente dans la formation des réseaux de neurones profonds et autres tâches d'apprentissage automatique. Le domaine émergeant de la *programmation différenciable* a récemment réussi à généraliser l'apprentissage profond en appliquant une optimisation à base de gradient via la DA à des applications de plus en plus sophistiquées et diversifiées.

Cependant, les outils les plus populaires pour la DA sont hyper-specialisés pour les taches d'apprentissage profonde. La performance de la DA elle-même et du code differencié resultant souffre dès que ses applications se détournent des reseaux de neurones profonds. Les méthodes de DA les plus populaires effectuent également leure différenciation pendant l'éxécution, ce qui entraîne une surcharge d'exécution avec, pour chaque différenciation, des millions d'étapes nécessaire.

Certains de ces problems peuvent être atténués en utilisant la DA à l'avance dans un compilateur. Cependant, les méthodes existantes fonctionnent principalement sur des representations intermédiaires (RIs) de compilateur de bas niveau qui perdent le context et les information par rapport au programme d'origine. De plus, la forme la plus courante de la DA entre entraîne un coût de mémoire asymptotiquement élevé par rapport au program d'origine, que la procedure de la DA soit effectuée au moment de la compilation ou de l'exécution.

Pour relever ces défits, cette these introduit LAGrad, un système de la DA en mode inverse fonctionnant au moment de la compilation qui exploite les informations de haut niveau dans MLIR pour créer un code différencié plus efficace. LAGrad utilise un collection de nouvelles optimisiations statiques qui profitent de la sémantique des dialects MLIR de haut niveau pour exploiter la rareté et le flux de contrôle structuré du code généré. En

utilisant ceci, LAGrad est capable d'accélérer la vitesse 2.8x en utilisant 35x moins de mémoire, comparé aux systems de la DA utilisés dans l'apprentissages automatiques et la vision par ordinateur dans le monde réel. LAGrad est le premier outil connu par l'auteur à exploiter la structure et la rareté présente dans la DA à l'aide d'optimisation statique dans un compilateur.

Acknowledgements

I would like to extend my deepest gratitude to my advisor, Professor Christophe Dubach, for his wisdom and continued support throughout my Master's degree. His guidance was instrumental in shaping me into the researcher that I am today, and his emphasis on accessible teaching, navigating the research landscape, and technical excellence are lessons that I will take with me for life.

I would also like to thank Ivan R. Ivanov for allowing me to edit and submit my CC'23 paper in his hotel room during the 2022 US LLVM Developers' Meeting, which was critical for that paper's success. He, along with the wonderful people I have met while contributing to the Enzyme project (who are too numerous to name), played a key role in expanding my view of what life as a researcher could be like.

I would like to thank my cherished friends, family, Effusion A Cappella, and my group mates at the Compiler and Accelerator Synthesis Lab at McGill for making me feel at home in a new city. In particular, thank you to Celia Hameury for helping translate the abstract of this thesis into French. Finally, thank you to Nishat, for continually reminding me of the beauty and fullness of life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The widespread adoption of deep learning, paired with the increasing sophistication of machine learning (ML) models, has led to a growing interest in using gradient-based optimization to learn parameterized *differentiable programs*. This new paradigm, known as *differentiable programming*, generalizes deep learning to allow ML practitioners to write increasingly expressive models that combine gradient-based training with domain specific knowledge. Differentiable programming has already seen success in applications such as physics simulations [1], ray tracing [3], and many other fields [2, 4, 5].

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 [8, 9]. 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.

1.1 The AD Landscape

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 [10–13], and this the approach taken in this thesis.

Forward-mode AD augments each step of the input program with *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, the majority of this work focuses on reverse-mode AD.

Reverse-mode AD involves analyzing or 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 [13–15].

Finally, AD can be applied on languages of dramatically different abstraction levels. Popular ML frameworks such as PyTorch [16], JAX [17], and TensorFlow [18] perform AD on high-level multidimensional arrays. AD systems built into language-specific compilers and libraries become tied to their respective languages, sacrificing generality. In contrast, Enzyme [11] differentiates at the low-level 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 generated may hinder opportunity for novel, AD-specific optimizations.

1.2 Contributions

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 thesis, LAGrad applies several optimizations such as tape 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 [11], the current state-

of-the-art system, and up to 1400× speedup relative to PyTorch [16], 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 thesis are:

- LAGrad, an MLIR-based AD system that demonstrates 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 [16] and Enzyme [11] on a standard AD benchmark suite [9] that shows it outperforms the state of the art performance for both run time and memory consumption.

Authors' Contributions. The contributions of this thesis are published in the following paper:

- 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.*

Mai Jacob Peng performed the implementation, experimental evaluation, and writing of the paper. Christophe Dubach provided guidance for both the research direction and structure of the paper, in addition to editing the introduction of the paper.

Chapter-based Contribution. Every chapter of this thesis was composed and written by Mai Jacob Peng. Christophe Dubach provided feedback, guidance, and supervision on every chapter.

1.3 Thesis Organization

The remainder of the thesis is organized as follows.

- Chapter 2 includes necessary prerequisite information on Automatic Differentiation and tensor MLIR, establishing context for the rest of the work. It also covers related work in the field of AD and discusses the limitations of other approaches.
- Chapter 3 discusses unique characteristics of LAGrad's AD implementation due to the semantics of MLIR, in addition to a discussion of tape size reduction.
- Chapter 4 describes the static optimizations employed after the AD process is completed.
- Chapter 5 evaluates the efficacy of LAGrad's optimizations both individually and against existing state-of-the-art methods.
- Chapter 6 concludes with a discussion of limitations and opportunities for future work in the field of optimizing AD and differentiable programming.

Chapter 2

Background and Related Work

To contextualize the contribution of this thesis, we will discuss both the necessary prerequisite concepts and the recent work that has been done within the field of efficient differentiable programming.

2.1 Background

This section begins by covering the fundamental algorithm of automatic differentiation, with details on both forward-mode and reverse-mode AD. It then provides an overview of Multi-Level Intermediate Representation (MLIR) [19] that is followed by a discussion of the specific technical aspects of MLIR that are employed by this thesis.

2.1.1 Automatic Differentiation

Given a program that computes a function $y = f(x)$, the goal of automatic differentiation 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*.

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 is broken down into multiple simple operations, each of which produces a z_i intermediate value.

The two main methods for AD, forward and reverse, differ in the order that the partial derivative expressions in the chain rule are evaluated. We will see this concretely in a motivating example. Suppose we have the following function:

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

Our goal is to compute the derivatives of y with respect to both w and b. We begin by breaking down the function into small pieces and explicitly naming each intermediate value:

Primal

$$z = wx + b$$

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

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

We will now examine how to use both forward and reverse mode to compute the desired derivatives.

2.1.2 Forward Mode

Forward mode (or tangent-mode) AD requires fixing the gradient of each of a function's input x, then computing a *dual number* for each intermediate value z that stores the derivative of that value with respect to the input, $\dot{z} = \frac{\partial z}{\partial x}$. In our example, the process is initialized with $\dot{w} = \frac{\partial w}{\partial w} = 1$ and $\dot{b} = \frac{\partial b}{\partial w} = 0$. Observe that the fixing of dual numbers implies that forward mode AD only computes the derivative with respect to one function input at a time by settings its dual to 1, while setting all others to 0. Since we desire the derivative with respect to multiple inputs, the forward mode procedure must be run once for each input with the appropriate initial dual number configuration.

Once initial dual numbers are fixed, the chain rule is traversed from the input to the output by executing dual (also known as *tangent*) instructions immediately after their primal counterparts. This mirrors the evaluation order of the original program:

Primal	**Tangent (forward)**
$z = wx + b$	$\dot{z} = \dot{w}x + \dot{b}$
$\sigma = 1 + e^{-z}$	$\dot{\sigma} = -\dot{z}e^{-z}$
$y = \frac{1}{\sigma}$	$\dot{y} = \dot{\sigma}\frac{1}{\sigma^2}$

Each primal operation is augmented with a *pushforward* to compute its tangent. The combined execution of the primal and tangent computations is known as a *forward sweep*. Upon completion, the value $\dot{y}$ now contains the desired $\frac{\partial y}{\partial w}$. A second forward sweep is performed as outlined above to compute $\frac{\partial y}{\partial b}$, only with $\dot{b} = 1$ and $\dot{w} = 0$. Note that the tangent computation requires values that are computed as a result of primal operations, meaning the primal and tangent computations must be performed in tandem to ensure these

values are available to the tangent operations.

In general, forward mode AD requires as many forward sweeps as there are active input variables to compute gradient vectors. However, it is able to compute derivatives of a single input with respect to multiple outputs in a single forward sweep.

In machine learning applications, models typically have millions of trainable parameters. This results in gradient vectors, as required by gradient-based optimization, requiring millions of forward sweeps to compute.

2.1.3 Reverse Mode

In contrast to forward mode, reverse mode AD begins by fixing the derivative with respect to the function's output, then traversing the chain rule backwards from output to input. Each differentiated version of a primal operation is known as a *pullback* [10]. In contrast to dual numbers in forward mode, the pullback for the operation that produces z_i is a function that takes the propagated *gradient signal*, $\overline{z_i} = \frac{dy}{dz_n} \dots \frac{dz_{i+1}}{dz_i}$, and the input(s) z_{i-1} to output $\overline{z_{i-1}}$. This can be seen below:

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

As seen above, 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

$$\mathbf{J} \in \mathbb{R}^{m \times n} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Figure 2.1: The general Jacobian matrix for a function $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$.

must execute completely 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* [10]. The tape is a fundamental data structure of reverse-mode AD and incurs a memory overhead that does not exist in the original program. A summary of this example showing the results of both forward and reverse methods is shown below in Figure 2.2.

When $\mathbf{x}$ and $\mathbf{y}$ are vectors, AD can be used to compute the *Jacobian matrix*. The Jacobian is a matrix where each element is a partial derivative of one output element y_i with respect to one input element x_j. This is visualized in Figure 2.1.

Reverse-mode AD computes the gradient of one element of $\mathbf{y}$ with respect to all elements of $\mathbf{x}$ (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).

Primal	Tangent (forward)	Adjoint (reverse)
$z = wx + b$	$\dot{z} = \dot{w}x + \dot{b}$	$\overline{\sigma} = -(\overline{y})\frac{1}{\sigma^2}$
$\sigma = 1 + e^{-z}$	$\dot{\sigma} = -\dot{z}e^{-z}$	$\overline{z} = -(\overline{\sigma})e^{-z}$
$y = \frac{1}{\sigma}$	$\dot{y} = \dot{\sigma}\frac{1}{\sigma^2}$	$\overline{w} = \overline{z}x$

Figure 2.2: An example of computing $\overline{w} = \dot{y} = \frac{dy}{dw}$ using forward- and reverse-mode AD. Both forward and reverse mode precisely compute the desired value, but differ in the order that they traverse the chain rule.

2.2 Multi-Level Intermediate Representation (MLIR)

Multi-Level Intermediate Representation (MLIR) [19] is a compiler infrastructure and intermediate representation (IR) designed to be arbitrarily extensible. It is broadly grouped into *dialects*, which are collections of conceptually related operations and types. Different dialects express programs at dramatically varying levels of abstraction. For instance, the `linalg` dialect contains operations that express high-level linear algebra computations such as matrix multiplication and convolution. The `scf`, or *structured control flow* dialect, contains operations such as if-statements and for-loops. In contrast, the `cf`, or control flow dialect expresses unstructured control flow primitives such as conditional and unconditional branches to basic blocks. On the lower levels of abstraction that MLIR supports, the `LLVM` dialect closely mirrors the instruction set of LLVM IR with operations such as `getelementptr`.

This thesis primarily focuses on the high level `linalg`, `tensor`, and `scf` dialects of MLIR. The reason for this is these dialects offer semantic guarantees and high-level information that is unavailable in lower level IRs.

2.2.1 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 involving memory and mutation.

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.

For instance, the following `linalg.generic` operation computes a dot product:

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

```
13   } -> tensor<f32>
```

Listing 2.1: A dot product expressed as a `linalg.generic` op

It contains the following:

- Any number of tensor inputs and outputs (lines 7-8). Inputs provide data to the operation, while outputs define the shape of each result and initial values in the case of reductions.
- An indexing map for each input and output, which describes how to index into each tensor from the induction variables of each loop (lines 2-5).
- An iterator type for each loop, which explicitly define which loops correspond to reductions of the outputs (line 6).
- A body, which is a basic block that defines what operations must be performed for each iteration of the innermost loop (lines 9-12).

For simplicity, this thesis makes use of a Tensor Comprehensions (TC) [6] style notation to compactly show `linalg.generic` ops. The above dot product can be expressed with the following TC notation:

```
1 C[] += A[d0] * B[d0]
```

This conceptually represents the following pseudocode:

```
1 for d0 from 0 to <inferred d0 bound>:
2   C[] = C[] + (A[d0] * B[d0])
```

By modifying the indexing maps and iterator types, a `linalg.generic` op can be used to express a wide variety of computation over parallel loop nests. The following op represents a matrix multiplication:

```
1  %matmul = linalg.generic
2    {  indexing_maps = [
3         affine_map<(d0, d1, d2) -> (d0, d2)>,
4         affine_map<(d0, d1, d2) -> (d2, d1)>,
5         affine_map<(d0, d1, d2) -> (d0, d1)> ],
6      iterator_types = ["parallel", "parallel", "reduction"] }
7    ins(%A, %B : tensor<?x?xf32>, tensor<?x?xf32>)
8    outs(%C : tensor<?x?xf32>) {
9      ^bb0(%arg0: f32, %arg1: f32, %arg2: f32):
10       %0 = arith.mulf %arg0, %arg1 : f32
11       %1 = arith.addf %arg2, %0 : f32
12       linalg.yield %1 : f32 } -> tensor<?x?xf32>
```

Listing 2.2: Matrix multiplication as a `linalg.generic` op in MLIR

Observe the similarities between Listing 2.1 and Listing 2.2. Apart from the shapes of inputs and outputs, the only differences required to change the dot product to a matrix multiplication are the modified indexing maps (Lines 2-5) and iterator types (Line 6). The operations are otherwise completely identical. The equivalent TC-style representation of the matrix multiplication is:

```
C[d0, d1] += A[d0, d2] * B[d2, d1]
```

Which corresponds to the following pseudocode:

```
for d0 from 0 to <inferred d0 bound>:
  for d1 from 0 to <inferred d1 bound>:
    for d2 from 0 to <inferred d2 bound>:
      C[d0, d1] = C[d0, d1] + (A[d0, d2] * B[d2, d1])
```

In general, 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 has the following pseudocode representation:

```
for d0 from 0 to <inferred d0 bound>:
  ...
    for dm from 0 to <inferred dm bound>:
      Out[map_O] = f(T_1[map_1], ..., 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.

2.3 Related Work

After establishing the necessary prerequisite concepts of automatic differentiation and MLIR, we turn to related work in the field of efficient differentiable programming. Our discussion of related work begins with established work in the general field of compiler optimizations and the precise relation of this work to the field of differentiable programming. We then turn to alternative, specific methods of performing AD and discuss challenges faced by these existing methods that this thesis will address.

2.3.1 Related Compiler Optimizations

Slice analysis and activity analysis. In automatic differentiation, there are several cases where the computation of derivatives with respect to certain program values is not required. Given the function $f(x, y) = x$, the derivative $\frac{\partial f}{\partial y}$ is trivially zero. Another instance is when only some function inputs are considered trainable parameters and thus require gradients. In neural networks, the gradient with respect to the data is well-defined, but not practically useful for training. Computing the gradients with respect to data is thus an instance of unnecessary work.

To reduce the amount of unnecessary computation in AD, *activity analysis* is a static analysis that determines which values are relevant to the desired derivatives. Existing formulations of activity analysis [7, 11] operate on languages with mutation and pointer

aliasing, resulting in highly complex analyses to identify values that do not require derivative computation.

However, a much simpler static analysis is *slice analysis* [20], which determines the set of program statements (a *program slice*) that may affect a given program value. Since tensor MLIR is free from mutation and pointer aliasing, LAGrad employs a version of activity analysis that is equivalent to computing a program slice for each value. This approach results in a much simpler analysis than existing methods. A full discussion is presented later in subsection 3.3.1.

Persistent data structures. While the previous paragraph discusses an advantage of tensor MLIR's lack of mutation, there are also important drawbacks with respect to performance. Recall from subsection 2.2.1 that inserting an element into a tensor requires returning a copy to avoid mutating the original tensor. This general pattern, known as *copy-on-write*, introduces considerable overhead when compared to an in-place update. We will later see this overhead in the evaluation.

The general problem of optimizing the updates of immutable data structures is exactly one of the motivations for *persistent data structures* [21]. These are common in functional programming languages with immutable data structures. Persistent data structures record additional metadata to support the same copy-on-write semantics without the need to create deep copies at every mutation point. However, they create partial copies of data and introduce both implementation complexity and run-time overhead. For these reasons, LAGrad employs an alternate heuristic strategy discussed in section 4.1 to optimize the copy-on-write behaviour of tensors. This alternate method produces in-place updates with no run-time overhead when it is valid to do so.

2.3.2 Operator Overloading

The most commonly used methods of automatic differentiation in machine learning are based on operator overloading (OO). These include PyTorch [16], TensorFlow eager [18], Autograd [22], and JAX [17]. 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 creates programs that scale linearly in size with respect to the number of loop iterations, making compiler optimizations impractical on these representations [10].

A consequence of the larger program size of OO methods is that the overhead of any transformation on the program scales linearly with respect to that program's size. AD is one such example of a program transformation, but recent trends towards Just-In-Time (JIT) compilation within systems such as JAX have seen increased JIT compilation overhead as a direct result of OO-based tracing methods [1].

As we will see, retaining structured control flow when differentiating aids in reasoning about and optimizing the auxiliary data structures when computing adjoints. In addition, the large program size from unrolled loops adds a nontrivial compilation overhead even outside of the context of AD.

2.3.3 Source Transformation

Zygote [10] 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. This hurts the reusability of optimizations implemented for language-specific approaches and does not have

[1] `https://jax.readthedocs.io/en/latest/faq.html#jit-decorated-function-is-very-slow-to-compile`

the potential for bringing in generic optimizations such as those present in MLIR.

Tapenade [23] supports both forward and reverse mode AD on Fortran and C code. It, along with Zygote, operate on compiler intermediate representations that lose the structured control flow present in programs. This improves the system's generality by operating on programs without structured control flow, at the expense of losing this information and hindering the application of aggressive optimizations that exploit this structured control flow. Additionally, the Fortran and C languages reason about arrays as pointers, which dramatically increases the complexity of incorporating sparsity-based optimizations. This is due to how sparsity-based optimizations greatly rely on static rank and shape information of arrays.

Tangent [24] performs source-to-source AD on high level code, which is a similarity with the approach taken in this work. It boasts improved debugging by generating readable Python code, but is not built with performance as a goal. It incurs interpreter overhead for both primal and adjoint code, which presents challenges in performance critical ML workloads.

Fsmooth [12] 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. By not supporting reverse mode, 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 will show, this reduces the complexity of performing AD-specific static analyses and optimizations.

Chapter 3

Automatic Differentiation on MLIR Tensors

The next two chapters build upon the background and prior work of Chapter 2 by describing the unique approach taken by this thesis. This chapter outlines an overview of this work within the context of the broader automatic differentiation landscape, then details the contributions of this work that occur *during* the AD procedure. This includes both 1) the differentiation procedure for high level operations in MLIR, and 2) *tape size reduction*, an optimization that is performed during AD. The next chapter continues the technical contribution of this thesis by describing the optimizations that occur *after* the AD procedure.

3.1 Overview

This thesis proposes performing AD at the tensor MLIR [19] level, striking a middle ground between high-level operator-overloading libraries and low level LLVM IR first discussed in

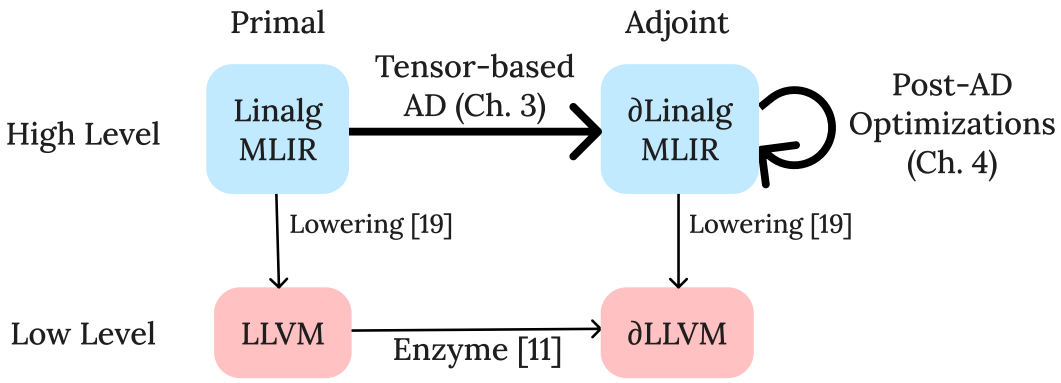

Figure 3.1: An overview of the approach taken in this work. The path of lowering before AD is the current state-of-the-art approach. The alternate path denoted by bolded arrows is taken by LAGrad, the contribution of this work.

section 1.1 and later expanded in section 2.3. An overview of this work is outlined in Figure 3.1. This chapter 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 [25], 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 operations supported by LAGrad consist of basic mathematical operations (*e.g.,* `arith.addf`, `math.tanh`), if-statements (`scf.if`), and looping constructs (*e.g.,* `scf.for`, `linalg.generic`).

Operation	Pushforward	Pullback
$z = a + b$	$\dot{z} = \dot{a} + \dot{b}$	$(\overline{z}, \overline{z})$
$z = a \times b$	$\dot{z} = \dot{a}b + a\dot{b}$	$(\overline{z}b, \overline{z}a)$
$z = \exp(a)$	$\dot{z} = \dot{a}\exp(a)$	$(\overline{z}\exp(a))$
$z = \sin(a)$	$\dot{z} = \dot{a}\cos(a)$	$(\overline{z}\cos(a))$

Table 3.1: Pushforwards and pullbacks of elementary operations.

3.2 Differentiating Operators in MLIR

The discussion of differentiating MLIR programs will begin with differentiating linear sequences of elementary operations without control flow, then move on to the special handling required for each supported control flow construct.

In the absence of control flow, AD simply follows the process outlined in subsection 2.1.1. Forward mode involves augmenting each function with a pushforward that propagates tangent information through the computation, while reverse mode traverses the primal operations in reverse order and appends pullbacks to the adjoint program. In both instances, the conversion from mathematical concepts to MLIR code is straightforward. This is demonstrated in Figure 3.2, which converts the previously seen Figure 2.2 from math to MLIR. Pushforwards and pullbacks are registered for each supported op, with some examples detailed in Table 3.1.

Basic mathematical MLIR operations like `arith.addf` and `math.tanh` 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.

```
// z = wx + b
%wx = arith.mulf %w, %x
%z = arith.addf %wx, %b

// sigma = 1 + e^(-z)
%negz = arith.negf %z
%exp = math.exp %negz
%sigma = arith.addf 1, %exp
// y = 1 / sigma
%y = arith.divf 1, %sigma
```

```
// dsigma = -dy / sigma^2
%sigmasq = arith.mulf %sigma, %sigma
%ndy = arith.negf %dy
%dsigma = arith.divf %ndy, %sigmasq
// dz = -dsigma * exp(-z)
%ndsigma = arith.negf %dsigma
%exp = math.exp %negz
%dz = arith.mulf %ndsigma, %exp
// dw = dz * x
%dw = arith.mulf %dz, %x
```

Figure 3.2: The primal (left) and reverse-mode adjoint (right) from Figure 2.2, expressed in MLIR.

3.2.1 scf.if

Like if-expressions in high level languages, `scf.if` ops in MLIR are composed of a boolean predicate, a `then` block that executes if the predicate is true, and an `else` block that executes if the predicate is false. Due to the side-effect free nature of tensor MLIR, `scf.if` ops are expressions that produce values, not statements. This implies that an `scf.if` op must contain an `else` block to be valid.

Reverse Mode. To produce adjoints for 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. Immutable semantics generally guarantee this, though LAGrad will explicitly store condition values in memory to access them later if required (this is further discussed later with the *tape* in Subsection 3.2.2). LAGrad then collects *free values*, which are values used inside a block that are defined outside of that block. For each free value that appears in either branch of the `scf.if` op, a corresponding adjoint `scf.if` op is produced by recursively differentiating both then `then` and `else` blocks.

This mathematically corresponds to the following piecewise derivative rule:

$$z = \begin{cases} f(x) & \text{if } p \\ g(x) & \text{otherwise} \end{cases} \implies \overline{x} = \begin{cases} \overline{z}\frac{\partial f(x)}{\partial x} & \text{if } p \\ \overline{z}\frac{\partial g(x)}{\partial x} & \text{otherwise} \end{cases} \tag{3.1}$$

For example, consider the following primal `scf.if` op, which contains an SSA value `%x` has been previously defined and is thus *free* in the context of the `if` op.

```
1 %res = scf.if %p -> f64 {
2   scf.yield %x : f64
3 } else {
4   %0 = arith.mulf %x, 2 : f64
5   scf.yield %0 : f64 }
```

Listing 3.1: Example primal `scf.if` op.

The resulting pullback for `%x` is simply a mechanical translation of the above piecewise derivative rule shown in Equation 3.1:

```
1 %dres = ... // incoming gradient
2 %adj = scf.if %p -> f64 {
3   scf.yield %dres : f64
4 } else {
5   %d0 = arith.mulf %dres, 2 : f64
6   scf.yield %d0 : f64
```

Listing 3.2: Pullback of Listing 3.1

Observe from line 2 of Listing 3.2 that the same condition value is used in both primal and pullback (`%p`). The `then` block of the pullback (Listing 3.2, Line 3) differentiates the `then` block of the primal (Listing 3.1, Line 2) and vice versa with the `else` block.

Forward Mode. Like reverse mode, forward mode differentiation requires that the same branches are taken in both primal and tangent, meaning the same condition values must be

used. The pushforward for `scf.if` ops look quite similar to the pullback in reverse mode:

$$z = \begin{cases} f(x) & \text{if } p \\ g(x) & \text{otherwise} \end{cases} \implies \dot{z} = \begin{cases} \dot{x}\frac{\partial f(x)}{\partial x} & \text{if } p \\ \dot{x}\frac{\partial g(x)}{\partial x} & \text{otherwise} \end{cases}$$

However, forward mode computes primal operations in tandem with their pushforwards instead of at the end of the primal. This conceptually results in *augmenting* the primal `scf.if` op such that the primal and tangent operations are computed together:

```
%res, %dres = scf.if %p -> (f64, f64) {
  scf.yield %x, %dx : f64, f64
} else {
  %0 = arith.mulf %x, 2 : f64
  %d0 = arith.mulf %dx, 2 :  f64
  scf.yield %0, %d0 : f64, f64
}
```

Listing 3.3: Augmented primal and tangent (highlighted) of Listing 3.1.

3.2.2 scf.for

Reverse Mode. 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.

This is described below in Listings 3.4 and 3.5, where x describes any free values that appear in the loop while z_i describes any loop-carried values.

```
1 z_0 = initial_value
2 for i from lb to ub:
3   z_{i+1} = f(z_i, x)
```

Listing 3.4: A primal loop.

```
1 for i from reversed(lb to ub):
2   x̄ += z̄_i ∂f(z_i,x)/∂x
3   z̄_{i-1} = z̄_i ∂f(z_i,x)/∂z_i
```

Listing 3.5: Its corresponding adjoint.

Observe that the body of Listing 3.5 contains derivative expressions that potentially depend on z_i for all iterations of the primal loop. However, z_i may be overwritten during the course of the primal loop, making it inaccessible during the adjoint execution.

For this reason, loop-carried values that are overwritten with every subsequent iteration (represented by z_i) must be recorded to memory in a data structure known as the *tape* [7,10]. This results in the following generalized pattern:

```
1  tape = initialize a stack
2  z_0 = initial_value
3  for i from lb to ub:
4    tape.push(z_i)
5    z_{i+1} = f(z_i, x)
6
7  for i from reversed(lb to ub):
8    z_i = tape.pop()
9    x̄ += z̄_i ∂f(z_i,x)/∂x
10   z̄_{i-1} = z̄_i ∂f(z_i,x)/∂z_i
```

In general, the primal loop must execute to fully populate the z_i values required in the adjoint. This is consistent with how the primal computation must fully execute prior to the adjoint when differentiating linear sequences of instructions.

The tape introduces a memory overhead that is avoidable in certain contexts, which is later discussed in Section 3.4.

Forward Mode. Unlike reverse mode, forward mode absolves the need for the gradient tape because of how primal and tangent ops are computed together. This guarantees that intermediate values required by tangent computations will always be accessible:

```
z_0, ż_0 = initial_values
for i from lb to ub:
  z_{i+1} = f(z_i, x)
  ż_{i+1} = ż_i ∂f(z_i,x)/∂x
```

While reverse mode requires the creation of a new loop, forward mode reuses the primal loop by merely augmenting the loop body with tangent operations.

3.2.3 linalg.generic

Recall from subsection 2.2.1 that `linalg.generic` ops represent an abstraction over arbitrary computation within parallel loop nests.

Reverse Mode. 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 pseudocode representation of a `linalg.generic` op, first discussed in subsection 2.2.1:

```
for d0 from 0 to <inferred d0 bound>:
  ...
    for dm from 0 to <inferred dm bound>:
      Out[map_O] = f(T_1[map_1], ..., 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 indexing map 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>:
```

```
%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>
```

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

(a) The `linalg.generic` matrix multiply.

```
%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>
```

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

(b) The pullback with respect to `%A`.

Figure 3.3: A matrix multiplication and its corresponding pullback in MLIR (and simplified TC notation [6]). The differences as a result of the AD process are highlighted.

```
...
 for dm from 0 to <inferred dm bound>:
  T̄_i[map_i] = f'(T_1[map_1], ..., T_N[map_N], Ōut[map_O], T̄_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 3.3. 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*, meaning its value is not a direct data dependency of the adjoint. The reason for this is further discussed in section 3.4 after covering the tape in more detail. If this condition is not met, a fallback is to immediately lower the `generic` op to a nest of

`scf.for` ops and differentiate it as such.

Forward Mode. Rather than creating a distinct pullback op, forward AD creates an augmented pushforward that iterates in the same order as the primal:

```
for d0 from 0 to <inferred d0 bound>:
  ...
    for dm from 0 to <inferred dm bound>:
      Out[map_O] = f(T_1[map_1], ..., T_N[map_N], Out[map_O])
      Out_dot[map_O] = f'(T_1[map_1], T_1_dot[map_1] ..., T_N[map_N], T_N_dot[map_N], Out[map_O], Out_dot[map_O])
```

Each active input tensor T_i is augmented with a corresponding dual $\dot{T}_i$ that is assigned the same indexing map map_i. The iterator types of the augmented primal are the same as the original.

3.2.4 Differentiation Summary

We have now covered the differentiation rules for each of the supported complex control flow operators in MLIR. As we have seen, reverse mode has greater complexity than forward mode AD due to the additional overhead stemming from the reversed control flow and the tape. However, recall that this complexity is a worthwhile tradeoff because machine learning applications often require differentiating a scalar output with respect to many inputs, which reverse mode AD can complete with a single pass. Forward mode AD requires as many passes as inputs to complete the same task, but excels at differentiating functions with more outputs than inputs.

To maximize flexibility, LAGrad supports both modes of AD with optimizations that benefit both. Optimizations performed during AD are covered in section 3.4, while optimizations performed after AD are covered in chapter 4. Before discussing the former kind of optimizations, we will cover To Be Recorded analysis, an important analysis that is

common in AD systems. One component of To Be Recoded analysis is *effective use analysis*, which will be reused in the novel optimization discussed in section 3.4.

3.3 To Be Recorded Analysis in MLIR

As previously discussed in subsection 3.2.2, the reversed control flow of reverse-mode automatic differentiation implies that some values required in the adjoint are overwritten during the execution of the primal. These values must be recorded on the gradient tape to ensure they are accessible. However, not all values from the primal need to be written to the tape.

Consider Listing 3.6, where all primal values are recorded to the tape by default. Lines 2-9 make up the augmented primal, where Lines 3-4 allocate tapes for primal values and Lines 7-8 store their values across loop iterations. Lines 11 and onward make up the adjoint where $\overline{x}$ is computed:

```
1  func mul(x: f64, n: i64):
2    // Primal
3    x_tape = initialize empty stack
4    s_tape = initialize empty stack
5    s = 0
6    for i from 0 to n:
7      x_tape.push(x)
8      s_tape.push(s)
9      s = s + x
10
11   // Adjoint
12   ds = 1, dx = 0
13   for i from reversed(0 to n):
14     x = x_tape.pop()
15     s = s_tape.pop()
16     dx += ds
17   return dx
```

Listing 3.6: A primal and adjoint loop that stores all primal values to tapes.

```
1  func pow(x: f64, n: i64):
2    // Primal
3    x_tape = initialize empty stack
4    p_tape = initialize empty
      stack
5    for i from 0 to n:
6      x_tape.push(x)
7      p_tape.push(p)
8      p = p * x
9
10   // Adjoint
11   dp = 1, dx = 0
12   for i from reversed(0 to n):
13     x = x_tape.pop()
14     p = p_tape.pop()
15     dx += p * dp
16     dp = dp * x
17   return dx
```

(a) A naive adjoint implementation that records both x and p to the tape.

```
func pow(x: f64, n: i64):
  // Primal

  p_tape = initialize empty
   stack
  for i from 0 to n:

    p_tape.push(p)
    p = p * x

  // Adjoint
  dp = 1, dx = 0
  for i from reversed(0 to n):

    p = p_tape.pop()
    dx += p * dp
    dp = dp * x
  return dx
```

(b) An equivalent implementation that reads the value of x (Line 16) instead of storing it.

Figure 3.4: An example adjoint implementation where a value x is stored to the tape but never modified over the primal, making its storage unnecessary. The unnecessary tape instructions are highlighted.

Observe the adjoint loop (Lines 13-16). Lines 14 and 15 emit pops to the stored values of x and s, but these values are unused in the rest of the adjoint loop body. This is due to the generated pullback for the addition operator (Line 16), which does not have dependencies on either x or s. Thus, neither x nor s need to be written to the tape to the compute the adjoint.

As another example, consider Figure 3.4a. Due to the pullback for the multiplication operation (Lines 15 and 16), the values of both x and p are required in the adjoint.

However, observe that the value of x is never overwritten during the course of the program.

n copies of x are written to the tape during the primal execution (Line 6), but it is possible to simply use the initial value of x in the adjoint without changing the final computation of $\overline{x}$. To summarize, primal values need not be recorded to the tape if either 1) their values are not actually used in the adjoint, or 2) their values are directly accessible in the adjoint since they are not overwritten.

To this end, *To Be Recorded* (TBR) analysis [7] is a static analysis with the goal of determining which values must be recorded to the tape, such that unnecessary values are not recorded. The initial formulation of TBR analysis was implemented to work using C as the source language in the Tapenade AD system. This thesis builds on prior work [7] to extend TBR analysis to tensor MLIR, which has key semantic differences from C.

Like Tapenade, LAGrad implements TBR analysis in three stages: activity analysis, effective use analysis, and killed analysis. The results of activity analysis are used during effective use analysis, and the results of effective use analysis are reused in tape size reduction, one of the novel optimizations presented in this thesis. These three phases are each discussed in turn.

3.3.1 Activity Analysis

The goal of activity analysis is to determine the set of values within a program that can potentially influence the derivative computation. This is equivalent to finding the set of values v that require differential values $\overline{v}$ to be computed.

Users of AD systems specify a subset of function inputs $\mathcal{X}$ and function outputs $\mathcal{Y}$ that they wish to differentiate with respect to, yielding $\frac{\partial y}{\partial x}$ for all $x \in \mathcal{X}, y \in \mathcal{Y}$. First, *top-down* activity analysis is run to find all values v that depend on some $x \in \mathcal{X}$, then *bottom-up* activity analysis is run to find all values w such that some $y \in \mathcal{Y}$ depends on w. The final

active set is the set intersection of top-down and bottom-up active values.

If the input programs are free from memory side effects, the notion of dependence can be computed by traversing define-use and use-define chains, which is straightforward given the SSA nature of tensor MLIR [25]. However, with programs that contain side effects, there may be additional data dependencies that arise from reading from and writing to memory, adding to the complexity of activity analysis. Fortunately, tensor MLIR is free from such side-effects and thus activity analysis in LAGrad is able to elide much of the complexity present in other AD systems [7, 11].

3.3.2 Effective Use Analysis

The process of effective use analysis is to determine which primal values are required when computing the adjoint. Recall from Table 3.1 that the pullback of a multiplication $z = a \times b$ introduces a dependency on its primal operands $(\overline{z}b, \overline{z}a)$, while the pullback of an addition does not. Effective use analysis computes $AdjU$ sets: each value v has a corresponding set $AdjU(v)$, which is the set of primal values required to compute $\overline{v}$.

To compute $AdjU$ for a function, each operation e in the function has its corresponding $AdjU(e)$ computed per the rules in Table 3.2. Then, $AdjU$ sets are computed by traversing operations in a top-down data flow manner per the equations in Table 3.3, which account for control flow that may occur in the program in addition to propagating effective use information through sequential operations. Table 3.3 refers to $Kill$ sets, which are generally empty in tensor MLIR and further explained in subsection 3.3.3. Finally, the result of effective use analysis is the $AdjU(t)$ for each value t that is returned from the function.

Prior work [7] shows that $AdjU$ computation generally requires an iterative, fixed point algorithm on programs with unstructured control flow. However, the data flow equations

Operation (e)	**Resulting** $AdjU(e)$
`arith.addf` e_1, e_2 `arith.subf` e_1, e_2	$AdjU(e_1) \cup AdjU(e_2)$

`arith.mulf` e_1, e_2
`arith.divf` e_1, e_2
`linalg.dot ins(`e_1, e_2`)`
`linalg.matvec ins(`e_1, e_2`)`
`linalg.vecmat ins(`e_1, e_2`)`
`linalg.matmul ins(`e_1, e_2`)`
`linalg.batch_matmul ins(`e_1, e_2`)`
	(`if` e_1 active `then` $AdjU(e_1) \cup \{e_2\}) \bigcup$
	(`if` e_2 active `then` $AdjU(e_2) \cup \{e_1\}$)

$\log(e_1), \exp(e_1)$, $\sin(e_1), \cos(e_1)$, $\tanh(e_1), \ldots$	$AdjU(e_1) \cup \{e_1\}$
`tensor.insert` e_1 `into` e_2`[`e_3`]`	$AdjU(e_1) \cup AdjU(e_2) \cup \{e_3\}$
`tensor.extract` e_1`[`e_2`]`	$AdjU(e_1) \cup \{e_2\}$
`tensor.insert_slice` e_1 `into` e_2`[`e_3`][`e_4`][`e_5`]`	$AdjU(e_1) \cup AdjU(e_2) \cup \{e_3, e_4, e_5\}$
`tensor.extract_slice` e_1`[`e_2`][`e_3`][`e_4`]`	$AdjU(e_1) \cup \{e_2, e_3, e_4\}$

Table 3.2: *AdjU* construction rules for individual operations.

with structured control flow can be computed top-down in a single pass without requiring iterative methods.

3.3.3 Killed Analysis

The final stage of TBR analysis is *killed analysis*, where the aim is to find values that have been overwritten or would otherwise be inaccessible in the adjoint unless explicitly written to the tape. As MLIR is in SSA form, killed analysis is greatly simplified due to the inability to overwrite SSA values. As such, the only values that are overwritten in tensor MLIR with structure control flow are loop-carried iteration arguments and arguments that depend on them.

Control-flow construct (B)	**Data Flow Equation** $(AdjU(B))$
Sequential, $B = B_1 \to B_2$	$AdjU(B) = ((AdjU(B_1) \setminus Kill(B_2)) \cup AdjU(B_2))$
$B =$ scf.if p $\{B_1\}$ else $\{B_2\}$	$AdjU(B) = AdjU(B_1) \cup AdjU(B_2)$
$B =$ scf.for ... $\{B_1\}$	$AdjU(B) = AdjU(B_1)$

Table 3.3: Structured data-flow equations for $AdjU$ construction with control flow, as formulated by [7].

The final set of values resulting from TBR analysis is the intersection of loop iteration arguments with the $AdjU$ set of the function.

Full TBR Example. Consider the following program.

```
func @myfunc(%x: f64, %n: i64) -> f64 {
  %sf, %rf = scf.for %iv = 0 to %n iter_args(%s = 0.0, %r = 1.0) {
    %s_next = arith.addf %s, %x
    %r_next = arith.mulf %r, %x
    scf.yield %s_next, %r_next
  }
  %final = arith.addf %sf, %rf
  return %final
}
```

Suppose we want to differentiate with respect to `%x`. The first stage of TBR analysis is activity analysis. Top down activity analysis begins from `%x`, then propagates through its def-use chains to yield {`%x, %s_next, %r_next, %s, %r, %sf, %rf, %final`}. Bottom up activity analysis begins with `%final`, then propagates through its use-def chains to yield the same set. The intersection of top-down and bottom-up activity analyses are the same, thus all of those values are marked active.

We next consider effective use analysis. The process is initialized by setting $AdjU(\%v) = \emptyset$ for all SSA values $\%v$ in the program. We begin by computing $AdjU$ sets for primitive operations, which consist of `arith.mulf` and `arith.addf` operations. These correspond to

multiplication and addition, which have construction rules detailed in the first two rows of Table 3.2. From these rules, $AdjU($`%r_next`$) = \{$`%r, %x`$\}$, while the $AdjU$ sets for all other values remain empty. The data flow equations in Table 3.3 are then employed to propagate the $AdjU$ sets down to the final return value, which results in the final $AdjU$ set of $\{$`%r, %x`$\}$.

Finally, we consider killed analysis. The set of loop carried iteration arguments and values that depend on them is $\{$`%s, %r, %s_next, %r_next`$\}$. The intersection of $AdjU$ and killed values is $\{$`%r`$\}$, and thus `%r` is the only value that must be written to the tape to compute the adjoint. Such an adjoint is shown below:

```
func @myfunc(%x: f64, %n: i64) -> f64 {
  %tape = memref.alloc(%n) : memref<?xf64>
  %sf, %rf = scf.for %iv = 0 to %n iter_args(%s = 0.0, %r = 1.0) {
    memref.store %r, %tape[%iv]
    %s_next = arith.addf %s, %x
    %r_next = arith.mulf %r, %x
    scf.yield %s_next, %r_next
  }
  %final = arith.addf %sf, %rf

  %dfinal = 1.0
  %ds = %dfinal
  %dr_init = %dfinal
  %dx_init = 0.0
  scf.for %iv = %n to %0 iter_args(%dx = %dx_init, %dr = %dr_init) {
    %r = memref.load %tape[%iv]
    scf.yield %dx + (%r * %dr) + %ds, %x * %dr
  }
  return %dx
}
```

3.4 Tape Size Reduction

While TBR analysis aims to minimize the set of values that are written to the tape, it does not consider that values can sometimes be recomputed instead of stored. Recomputation of primal values generally alleviates the memory overhead of the gradient tape, but can result in asymptotically worse performance as a trade-off. However, there exist instances of differentiated programs where recomputation does not incur this asymptotically worse cost. LAGrad employs a novel static analysis that builds on prior work [7] with the aim of deciding whether to store or recompute primal values.

3.4.1 Motivating Examples

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.

Example 1. Consider Listing 3.7, which implements a simple function as an MLIR loop. Line 2 shows `%res_it`, a loop carried dependency that receives an initial value of 0.0, then is overwritten by the value yielded on Line 4 at the start of the next iteration.

A naive adjoint of this program is shown in Listing 3.8. It requires the primal value `%y` from all iterations of the primal loop, as shown by its usage in Lines 12-13. Observe that while the original program requires $\mathcal{O}(1)$ memory, the overhead of the tape requires $\mathcal{O}(n)$ memory where n is the number of loop iterations.

However, it is possible to recompute `%y` rather than writing it to the tape. This is shown in Listing 3.9, where `%y` is recomputed in the adjoint on Line 5 rather than being loaded from the tape. In addition to no longer requiring the tape, the primal loop is trivially

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

Listing (3.7) The primal function $f(x, n) = \sum_{i=0}^{n} xi^2$, expressed in MLIR.

```
func @grad_f_v1(%x: f32, %n: index) -> f32 {
  %tape = memref.alloc(%n) : memref<?xf32>
  // Primal loop
  %res = scf.for %iv=0 to %n iter_args(%res_it=0.0) {
    %y = %iv * %iv : f32
    memref.store %y, %tape[%iv]
    scf.yield %res_it + (%y * %x) : f32 }

  %dres = arith.constant 1.0 : f32
  // Adjoint loop
  %dx = scf.for %iv=%n-1 to -1 step -1 iter_args(%dx_it=0.0) {
    %y = memref.load %tape[%iv] : f32
    scf.yield %dx_it + %dres * %y : f32
  }
  return %dx : f32
}
```

Listing (3.8) A naive adjoint of Listing 3.7. The computation, storage, and usage of primal value %y is highlighted.

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

Listing (3.9) An optimized adjoint of Listing 3.7 that recomputes primal values rather than storing them to the tape. The primal loop is elided due to no longer being necessary. The recomputation of %y is highlighted.

Figure 3.5: A primal loop, naive adjoint, and optimized adjoint demonstrating the benefit of recomputing primal values.

elided through MLIR's dead code elimination because it is no longer required. Listing 3.9 demonstrates that choosing to recompute yields an adjoint that can be computed using $\mathcal{O}(1)$ memory, just like the primal in Listing 3.7. Furthermore, since the recomputation is relatively cheap, the overall runtime complexity of this adjoint is $\mathcal{O}(n)$, which is the same as the primal. In other words, recomputation allows for an asymptotic reduction in the memory usage of AD without compromising on runtime performance.

Example 2. However, the process of recomputing primal values is not always inexpensive. Consider Listing 3.10, in which the `%p` intermediate value is a loop carried dependency and thus requires its previous value at each iteration.

The corresponding pullback is shown in Listing 3.11, which again has an $\mathcal{O}(n)$ memory overhead from the tape, in contrast to the $\mathcal{O}(1)$ memory usage of the primal. To elide the tape, the value `%p` used in the adjoint loop must be recomputed instead of loaded from the tape. The first adjoint iteration requires the last value of `%p`, the second adjoint iteration requires the second last value, and so on. However, as nothing is cached on the tape and `%p` depends on its previous value, recomputing it requires starting from the very first primal iteration for every adjoint iteration.

This results in the adjoint shown in Listing 3.12. Though the adjoint now uses $\mathcal{O}(1)$ memory like the primal, the recomputation of `%p` results in a triangular lattice of computation across the now nested loops (Lines 3-4). This is visualized in Figure 3.7 and results in the adjoint program having a computational complexity of $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n)$ of both the original computation and the tape-based adjoint in Listing 3.11. In contrast to the previous example, recomputation incurs an asymptotically worse performance as a trade-off for saving on memory.

As we have seen, there are some loops where recomputing primal values instead of

```
func @g(%x: f32, %n: index) -> f32 {
  %r = scf.for %iv = 0 to %n iter_args(%p = 1.0) {
    scf.yield %p * x : f32
  }
  return %r : f32
}
```

Listing (3.10) The primal function $g(x, n) = x^n$

```
func @grad_g(%x: f32, %n: index) -> f32 {
  %tape = memref.alloc(%n) : memref<?xf32>
  // Primal loop
  %r = scf.for %iv = 0 to %n iter_args(%p = 1.0) {
    memref.store %p, %tape[%iv]
    scf.yield %p * x : f32
  }

  // Adjoint loop
  %dx, %dr = scf.for %iv = %n-1 to -1 step -1 iter_args(%dx_it=0.0, %dr_it
   =1.0) {
    %p = memref.load %tape[%iv]
    scf.yield %dx_it + %p * %dr_it, %dr_it * x
  }
  return %dx
}
```

Listing (3.11) The adjoint of Listing 3.10, using a tape.

```
func @grad_g_v2(%x: f32, %n: index) -> f32 {
  // Adjoint loop
  %dx, %dr = scf.for %iv = %n-1 to -1 step -1 iter_args(%dx_it=0.0, %dr_it
   =1.0) {
    %p = scf.for %jv = 0 to %iv iter_args(%p_it = 1.0) {
      scf.yield %p_it * x : f32
    }
    scf.yield %dx_it + %p * %dr_it, %dr_it * x
  }
  return %dx
}
```

Listing (3.12) The adjoint of Listing 3.10 that recomputes instead of using the tape. The nested loop in the adjoint results in $\mathcal{O}(n^2)$ computation.

Figure 3.6: A primal loop and two adjoints that demonstrate recomputing is not always beneficial.

$$
\begin{aligned}
i = n-1, \quad & p_{n-1} = x \times x \times \ldots \times x \times x \\
i = n-2, \quad & p_{n-2} = x \times x \times \ldots \times x \\
& \vdots \\
i = 2, \quad & p_2 = x \times x \times x \\
i = 1, \quad & p_1 = x \times x \\
i = 0, \quad & p_0 = x
\end{aligned}
$$

Figure 3.7: The triangular lattice of computation required by eliding the tape in Listing 3.11 to produce Listing 3.12.

storing them on the tape can improve memory usage without incurring worse computation overhead, while other loops result in asymptotically worse performance when recomputing primal values. This work introduces *tape size reduction*, a novel static analysis that detects when the resulting overhead of recomputation is not asymptotically worse than the original computation.

3.4.2 Tape Size Reduction Algorithm

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 [7].

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, 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 [10, 11]. 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 is why `linalg` ops cannot be differentiated when their output arguments are effectively used. When these output arguments are stored to the tape, they must be read during the adjoint in precisely the reverse order of the primal. This implies 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.

3.5 Summary

In this chapter, we have covered the procedure to differentiate high-level constructs in tensor MLIR with both elementary operations and structured control flow. We have also covered tape size reduction, an optimization that detects cases to improve the memory usage of reverse mode AD without incurring runtime overhead. We will now discusses static optimizations that are performed after the completion of the AD process.

Chapter 4

Post-AD Optimizations

Once adjoint code is generated, it can be optimized by the compiler prior to being bufferized and lowered to executable code. This chapter discusses optimizations performed immediately after AD is performed.

The optimizations discussed in this chapter are independent of automatic differentiation and thus could be applied outside of AD contexts, but programs generated by the AD process demonstrate particular characteristics that present opportunities for these optimizations to be beneficial.

4.1 In-place Bufferization

Recall that MLIR's default bufferization will allocate new memory on each tensor op to preserve the immutable semantics of tensors. As we will see, this process incurs a potentially large performance and memory overhead. To combat this, LAGrad contains a number of methods to statically determine when it is safe to lower these ops to in-place updates that avoid unnecessary allocations and copies.

4.1.1 Memory Conflicts and Dominance-based Heuristics

Consider the following code snippet, where Line 2 semantically creates a new tensor by setting `%t` at index `%idx` to `%val`, leaving all other elements equal to their original values:

```
1 func @insert(%val: f64, %t: tensor<1024xf64>, %idx: index) {
2   %t_new = tensor.insert %val into %t[%idx] : tensor<1024xf64>
3   // ...uses of %t_new, %t...
4 }
```

Tensors in MLIR semantically represent immutable, multidimensional arrays that intentionally have their underlying memory abstracted away. To produce code that can be executed, tensor-level code must be *bufferized* to code that contains explicit memory buffers. Multiple bufferization implementations are possible, meaning the underlying memory of tensors is not necessarily a straightforward, one-to-one mapping of the tensor's shape to a block of memory. One such instance of this is in sparse tensors, where a potentially large tensor is mapped to a sparse storage scheme [26].

By default, MLIR will produce the following code as a result of bufferization (noting that the types of input parameters have been converted).

```
1 func @insert(%val: f64, %t: memref<1024xf64>, %idx: index) {
2   %t_new = memref.alloc() : memref<1024xf64>
3   linalg.copy(%t, %t_new)
4   memref.store %val, %t_new[%idx] : memref<1024xf64>
5   // ...uses of %t_new, %t...
6 }
```

Line 2 allocates an entirely new memory buffer, line 3 copies over all elements of the original buffer into the newly allocated buffer, then line 4 updates the single desired element. Note that the `memref.store` is an explicit store to memory, while `tensor.insert` merely *describes* the creation of a new tensor and leaves the implementation up to a downstream lowering pass.

This default bufferization is an $\mathcal{O}(n)$ operation where n is the number of elements of the tensor, which is significantly more expensive than the $\mathcal{O}(1)$ operation of updating a single number in memory. However, consumers of `%t` expect to read its unmodified value, meaning the allocation and copy are required to ensure the correctness of the program.

Crucially, the `tensor.extract` of `%original` involves a use of `%t` that *postdominates* the `tensor.insert` to `%t`. When bufferizing an insert (or insert slice) to a tensor t, if there are no uses of t that postdominate the insert (resp. insert slice), the compiler can guarantee that it is safe to lower to an in-place update which avoids the extra allocation and copy.

LAGrad makes use of MLIR's existing dominance analysis to implement this. Given the following code, which no longer has a data hazard due to the lack of postdominating uses of `%t`:

```
func @insert(%val: f64, %t: tensor<1024xf64>, %idx: index) {
  %t_new = tensor.insert %val into %t[%idx] : tensor<1024xf64>
  // ...uses of %t_new...
  return
}
```

LAGrad will produce the following lowered code:

```
func @insert(%val: f64, %t: memref<1024xf64>, %idx: index) {
  memref.store %val, %t[%idx] : memref<1024xf64>
  // ...uses of %t_new are replaced with %t...
  return
}
```

There is an analogous heuristic that is applied to `tensor.extract_slice` ops, which extract a slice (*i.e.,* a portion, which is itself a smaller tensor) of a source tensor. Consider the following example, where a row of `%t` is extracted to make `%slice` on line 2:

```
func @extract_slice(%t: tensor<1024x1024xf64>, %idx: index) {
  %slice = tensor.extract_slice %t[%idx, 0] [1, 1024] [1, 1] :
   tensor<1024x1024xf64> to tensor<1024xf64>
```

```
3   // ...uses of %slice...
4   return
5 }
```

The semantics of the program require that `%slice` and `%t` are independent of each other, such that modifications to one will not affect the other. Default bufferization therefore inserts an extra memory allocation for the slice that is copied over, resulting the following:

```
1 func @extract_slice(%t: memref<1024x1024xf64>, %idx: index) {
2   %slice = memref.alloc() : memref<1024xf64>
3   %view = memref.subview %t[%idx, 0] [1, 1024] [1, 1] : memref<1024
     x1024xf64> to memref<1024xf64>
4   linalg.copy(%view, %slice)
5   // ...uses of %slice...
6   return
7 }
```

Lines 2 to 4 demonstrate the required allocate, subview, then copy pattern that is required to ensure the immutability of both the source and destination buffers. However, for a given extract slice of a source tensor t, it is safe to avoid an extraneous allocation and copy if there are no uses of t that postdominate the extract slice op. In particular, a postdominating use may mutate the underlying memory of t, violating the correctness of consumers of its slice. In the absence of such a use, LAGrad will produce the following lowered code:

```
1 func @extract_slice(%t: memref<1024x1024xf64>, %idx: index) {
2   %slice = memref.subview %t[%idx, 0] [1, 1024] [1, 1] : memref<1024
     x1024xf64> to memref<1024xf64>
3   // ...uses of %slice...
4   return
5 }
```

The same optimization is not required for `tensor.extract` ops because such ops produce a scalar SSA value. These SSA values will be lowered to registers that will not change even if the underlying memory of the source tensor is mutated.

```
1 %slice = tensor.extract_slice %A[2] : tensor<4x5xf64> to
     tensor<5xf64>
2 %updated = linalg.generic ... outs(%slice) ...
3 %result = tensor.insert_slice %updated into %A[2] : tensor<5
     xf64> into tensor<4x5xf64>
```

Listing 4.1: A tensor program that exhibits the read/compute/write pattern optimizable to an in-place update.

4.1.2 Insert-Extract Analysis

Consider Listing 4.1, where a slice of the tensor `%A` is read from (line 1), used to compute a new tensor of the same shape (line 2), then the new tensor slice is written back into `%A` at the same location (line 3). Our previous dominance heuristics will fail to efficiently bufferize this code because the insert slice of `%A` (line 3) postdominates the extract slice (line 1), resulting in a new buffer allocations for each op:

```
// Listing 4.1, Line 1: 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)
// Listing 4.1, Line 2: allocate + copy for linalg.generic
%updated = memref.alloc() : memref<5xf64>
linalg.copy(%slice, %updated)
linalg.generic ... outs(%updated) ...
// Listing 4.1, Line 3: allocate + 2 copies for tensor.insert_slice
%result = memref.alloc() : memref<4x5xf64>
linalg.copy(%A, %result)
%write_view = memref.subview %result[2] : memref<4x5xf64> to memref
   <5xf64>
linalg.copy(%updated, %write_view)
```

However, it is possible to bufferize the listing to an in-place update where the `linalg.generic` op updates the underlying memory of `%A` directly:

```
%slice = memref.subview %A[2] : memref<4x5xf64> to memref<5xf64>
```

```
linalg.generic ... outs(%slice) ...
```

This read/compute/write pattern is prevalent in the generated output of LAGrad due to the accumulation of gradient signals. As such, this work introduces *Insert-Extract (IE) Analysis* to find patterns where a slice of a tensor may be updated in place. The following conditions must be met:

- The destination of the extract slice must have exactly one use that postdominates the extract slice. That use must be in the destination of an insert slice, which is termed the *matching* insert slice.
- The matching insert slice must have the same indexing operands (offsets, sizes, and strides) as the extract slice.
- The destination of the insert slice must not have any uses that postdominate the insert slice.
- The result of the extract slice must appear as an output of a linalg.generic op, and the result of the linalg.generic op must be used as the input to the insert slice.

If IE Analysis determines these conditions to be met, it will group the paired extract/compute/insert ops and bufferize them to avoid any extraneous memory allocations.

4.1.3 Summary

Using a combination of more straightforward dominance-based heuristics and Insert-Extract Analysis, LAGrad is able to perform aggressive bufferizations that update memory in-place rather than creating extra allocations and copies. All of these are heuristics, meaning there

may be cases where in-place updates are valid that remain undetected, but these heuristics are designed to be conservatively correct when applied.

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 [7]. 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 4.2. 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 13, where the computation of $\overline{Y}$ involves 8 multiplications since both Y and $\overline{Z}$ are 2 by 2 arrays. Once computed, only one entry of $\overline{Y}$ is relevant to the gradient of the output, as shown on line 15.

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

1. At least 50% of the 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.

```python
def f(x: float):
  # Most elements of Y are constant, thus inactive
  Y = np.zeros((2, 2))
  Y[1, 0] = x
  Z = Y ** 2
  return Z.sum()

def grad_f(x: float, g: float):
  Y = np.zeros((2, 2))
  Y[1, 0] = x

  dZ = np.broadcast_to(g, (2, 2))
  dY = 2 * Y * dZ
  # Inactive elements of dY are unused, meaning their computation
   could have been avoided.
  dx = dY[1, 0]
  return dx
```

Listing 4.2: Primal and adjoint functions containing sparsely active arrays, where most elements are constant with respect to the function 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 4.1: Visualization of the sparsity of primal (top) and adjoint (bottom) values in Listing 4.2.

```
// 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 4.3: 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.

Note that the values in the example in Listing 4.2 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 [27]. This process automatically generates code to map iteration variables to the

indexing scheme of packed triangular storage. These methods are shown in Listing 4.3. The packing of lower triangular tensors potentially improves cache locality of computation operating on these tensors.

Concretely, the function shown in Listing 4.2 will have the following representation when expressed in MLIR:

```
func @f(%x: f64) -> f64 {
  %Y_space = linalg.init_tensor [2, 2] : tensor<2x2xf64>
  %zero = arith.constant 0.0 : f64
  %Y_init = linalg.fill(%zero, %Y_space) : f64, tensor<2x2xf64>
  %Y = tensor.insert %x into %Y_init[1, 0]
  %Z = math.powf %Y_init : tensor<2x2xf64>
  %sum = linalg.generic ... ins(%Z: tensor<2x2xf64>) {
  ^bb0(%in: f64, %out: f64):
    linalg.yield %in + %out : f64
  }
  return %sum : f64
}
```

To generate sparse code, one must annotate the tensor types in the program with their static sparsity as follows:

```
func @f(%x: f64) -> f64 {
  %Y_space = linalg.init_tensor [2, 2] : tensor<2x2xf64>
  %Y_packed = lagrad.pack %Y_space : tensor<2x2xf64> to tensor<2
   x2xf64, "pltri">
  %zero = arith.constant 0.0 : f64
  %Y_init = linalg.fill(%zero, %Y_packed) : f64, tensor<2x2xf64, "
   pltri">
  %Y = tensor.insert %x into %Y_init[1, 0]
  %Z = math.powf %Y_init : tensor<2x2xf64, "pltri">
  %sum = linalg.generic ... ins(%Z: tensor<2x2xf64, "pltri">) {
  ^bb0(%in: f64, %out: f64):
    linalg.yield %in + %out : f64
  }
  return %sum : f64
}
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ a & 0 & 0 & 0 \\ b & d & 0 & 0 \\ c & e & f & 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} a & b & c & d & e & f \end{bmatrix}$$

Figure 4.2: Conversion of lower triangular matrix (`tensor<4x4xf64, "pltri">`) to packed representation (`tensor<6xf64>`) in the 4×4 case. a to f represent nonzero scalar elements.

Note the addition of the `lagrad.pack` op, which denotes a conversion between an empty tensor and a packed representation (such as in Figure 4.2). This is necessary because the `linalg.init_tensor` op used to initialize new tensors (which ultimately lower to buffer allocations) does not support initializing tensors with nonstandard encodings.

The packing transformation converts annotated tensors to *column-major* packed representations, such that each operation that operates on a value with type `tensor<2x2xf64, "pltri">` is converted to have type `tensor<1xf64>`. In general, each packed tensor of rank r where the final two dimensions are of size d is converted to a tensor of rank $r - 1$ where the final two dimensions are replaced with a single dimension of size $\frac{d(d-1)}{2}$. This allows for conversion of batches of sparse matrices without additional handling.

Elementwise operations are converted by changing the type of the result, which is sufficient to change the semantics of the operation to skip over zero elements. This includes ops in the `arith` and `math` dialects, in addition to certain operations such as `linalg.fill`. For example, `arith.mulf %x : tensor<2x2xf64, "pltri">` becomes `arith.mulf %x : tensor<1xf64>`.

`linalg` operands are converted to loop nests with modified bounds that include explicit index computation to convert indices with respect to the fully materialized storage scheme into indices with the packed storage. An example of this is shown between versions 1 and 3

of Listing 4.3.

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 $\overline{v}$ in LAGrad has the same type as the primal value v. Thus, primal values that are manually marked lower triangular will have their gradients automatically annotated lower triangular, resulting in both primal and gradient computation benefiting from the optimization.

4.2.1 Summary

Recall from subsection 3.3.1 that activity analysis typically reasons about the activity of values in a coarse-grained manner, such that a single active element within a tensor implies the activity of that entire tensor. In contrast, Active Sparsity is a foray into a finer-grained notion of activity where only some parts of tensors are considered active. By looking at structured patterns, the overhead of tracking this finer-grained activity is constant with respect to the size of the tensors being tracked.

As we will see in the evaluation, this finer-grained tracking allows for reducing the required computation by half.

4.3 Adjoint Sparsity

In addition to active sparsity, another form of sparsity present in AD is when computing full Jacobian matrices. Recall that the process of computing a full 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

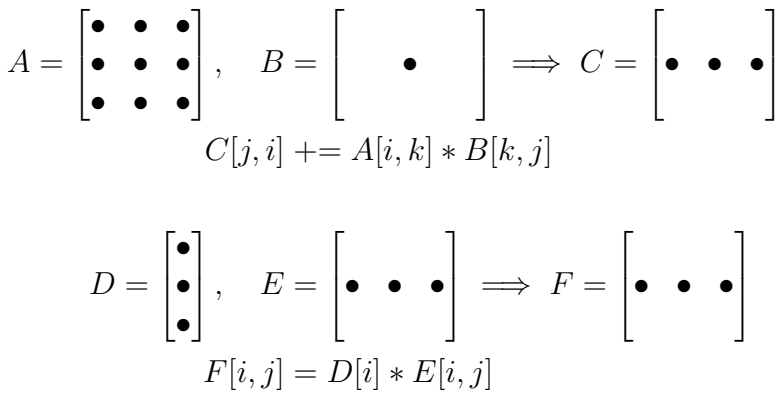

Figure 4.3: Examples of propagating per-dimension sparsity patterns through `linalg` ops. • represents a nonzero entry.

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.
- 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. These tensors are visualized in Figure 4.3, where the sparsity of C and F 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 F

despite its lack of inferred sparsity.

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 4.3, where the second argument B is sparse along both dimensions. This op has these indexing maps for B and C:

$$map_A = (d0, d1, d2) \to (d0, d2)$$
$$map_B = (d0, d1, d2) \to (d2, d1)$$
$$map_C = (d0, d1, d2) \to (d1, d0)$$

Note that A, being fully dense, does not contribute to the inferred sparsity of the output.

This results in the following:

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

The final result is that C has dimensions [sparse, dense], meaning it contains a nonzero row. This procedure is initialized with an explicit annotation on the seed vector of each differentiated function, then propagates top-down to maximize the sparsity found within each intermediate value.

4.3.1 Code Generation

Once the process of sparse propagation has completed, the resulting sparsity information is used to generate code that skips over sparse dimensions. Concretely, the above example is expressed as the following `linalg` op:

```
%result = linalg.generic {
  indexing_maps = [
    (d0, d1, d2) -> (d0, d2),
    (d0, d1, d2) -> (d2, d1),
    (d0, d1, d2) -> (d1, d0)],
  iterator_types = ["parallel", "parallel", "reduction"]
  }
  ins(%A, %B : tensor<?x?xf64>, tensor<?x?xf64>) outs(%C : tensor<?x
   ?xf64>) {
    ^bb0(%a: f64, %b: f64, %c: f64):
      linalg.yield %a * %b + %c
}
```

Without any modification from sparse propagation, the default lowering of this op would be to the following loop nest (with inferred loop bounds to iterate completely over their arguments):

```
for d0 from 0 to <inferred d0 bound>:
  for d1 from 0 to <inferred d1 bound>:
    for d2 from 0 to <inferred d2 bound>:
      C[d1, d0] += A[d0, d2] * B[d2, d1]
```

Sparse code generation results in omitting loop iterations from the inferred sparsity of the inputs to the `linalg` op. Recall that in our example, B is one-hot sparse along both dimensions, so $SparseDims(B) = \{d2, d1\}$. When iterating over sparse dimensions, only the nonzero index along those dimensions must be read. When a generated loop iterates over a one-hot dimension, it can be replaced as it will only read a nonzero value during *one* of its iterations. Thus, rather than generating the three nested loops, LAGrad emits only the loop for $d0$ after determining that the loops for $d1$ and $d2$ can be elided:

```
// within the compiler: d1, d2 = indices[B][1], indices[B][0]
for d0 from 0 to <inferred d0 bound>:
  C[d1, d0] += A[d0, d2] * B[d2, d1]
// within the compiler: indices[C][0] = d1
```

Listing 4.4: Code generated from a sparse `linalg` op

Note that an auxiliary compile-time data structure, `indices`, is used to record the location of nonzero values for each tensor. `indices` is a two-level nested dictionary *within the compiler* that maps each sparse dimension of each tensor to the indices of the nonzero values for that dimension. `indices` is populated by traversing the program in a top-down fashion at compile time, again making use of the predictable structured control flow of valid LAGrad programs. An example of this mapping is seen in Listing 4.4, where $d1$ and $d2$ are read from the `indices` data structure. After the execution of the lowered computation, `indices` is then updated

$$M = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}, \quad N = \begin{bmatrix} & \bullet & \\ & \bullet & \\ & \bullet & \end{bmatrix} \implies O = \begin{bmatrix} \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{bmatrix}$$
$$O[i,j] \mathrel{+}= M[i,k] * N[j,k]$$

Figure 4.4: A matrix multiplication with potential for adjoint sparsity optimizations in spite of producing a dense result.

to store the location of the nonzero row of C. However, `indices` is not materialized in the generated code.

If A has size $m \times k$ and B has size $k \times n$, this computes the correct value of C while reducing the computational complexity of the operation from $\mathcal{O}(mnk)$ to $\mathcal{O}(m)$.

We now turn our attention to another example in which the result of an operation is not sparse, but opportunities for sparse code generation are still present. Consider the operation shown in Figure 4.4, where $SparseDims(N) = \{d2\}$. A default lowering to loop nests would produce three triply-nested loops, exactly like the previous example:

```
for d0 from 0 to <inferred d0 bound>:
  for d1 from 0 to <inferred d1 bound>:
    for d2 from 0 to <inferred d2 bound>:
      O[d0, d1] += M[d0, d2] * N[d1, d2]
```

Due to the sparsity of N, the sparse code generation of LAGrad elides the innermost loop without changing the correctness of the result:

```
// within the compiler: d2 = indices[N][1]
for d0 from 0 to <inferred d0 bound>:
  for d1 from 0 to <inferred d1 bound>:
    O[d0, d1] += M[d0, d2] * N[d1, d2]
```

This sparsity-based optimization is possible even though the resulting O is fully dense. The density of O also implies that the `indices` data structure does not need to be updated.

Given that M has size $m \times k$ and N has size $n \times k$, the computational complexity of the operation is reduced from $\mathcal{O}(mnk)$ to $\mathcal{O}(mn)$.

4.3.2 Loop Nest Analysis

The high-level information available in `linalg` ops makes the process of sparse propagation straightforward. However, there are some nested loops that are not able to be represented using `linalg` ops due to restrictions in the IR. Consider the following loop:

```
func reduce_some(x, idxs, result):
  for i from 0 to 3:
    result[i] = x[idxs[i]]
```

This can be expressed equivalently in MLIR as a `linalg` op:

```
linalg.generic
  { indexing_maps = [(d0) -> (d0), (d0) -> (d0)],
    iterator_types = ["reduction"] }
  ins(%idxs)
  outs(%result) {
     ^bb0(%idx: index, %out: f64) {
       %xval = tensor.extract %x[%idx]
       linalg.yield %xval
     }
  }
```

In pseudocode, this produces the following adjoint:

```
func adj_reduce_some(x, indices, dresult):
  dx = zeros_like(x)
  for i from 3 to 0:
    dx[idxs[i]] = dresult[i]
```

Listing 4.5: A loop that does not write to every element of dx.

However, a problem arises when trying to express Listing 4.5 as a `linalg` op. `linalg` operations have the property of writing completely to their output operands, while the

loop above only writes to parts of dx. The selective updates follow an unpredictable pattern depending on the values in `indices`, which is only known at run time. Due to this, there is no indexing map that appropriately expresses the write pattern to `dx`, and thus the adjoint loop cannot be expressed as a `linalg` op. A fallback to produce a correct adjoint is to generate the loop as an `scf.for` op rather than a `linalg` op, but this approach is incompatible with sparse propagation analysis because the `scf.for` ops lack explicit indexing maps that make sparse propagation analysis possible.

To address this, this work introduces *loop nest analysis* to rediscover the information present in a `linalg` op (particularly the indexing maps) from a nest of `scf.for` ops. To this end, we define the term *well-behaved loop nest* to refer to loop nests that satisfy the following criteria:

1. The loop nest consists of an innermost loop and zero or more outer loops. Each outer loop contains exactly one loop directly within its body.

2. The operations in the innermost loop are `tensor.extract` ops, `tensor.insert` ops, or ops that operate on scalars. Well-behaved innermost loops do not contain operations that use tensors or other aggregates types.

3. All yielded operands in the innermost loop are the result of `tensor.insert` ops. All yielded operands in outer loops pass along the results of their direct child loop in the order they are given.

Well-behaved loop nests are somewhat like `linalg` ops, but can express some computations (such as Listing 4.5) that are impossible to express using only `linalg` ops. We will later see that they are useful when propagating sparsity when some information is not known at compile time. To further illustrate this concept, the following is an example

of a well-behaved loop nest per the analysis:

```
%res = scf.for i from 0 to M iter_args(%0 = ..., %1 = ...) {
  %res_0:2 = scf.for j from 0 to N iter_args(%2 = %0, %3 = %1) {
    %res_1:2 = scf.for k from 0 to K iter_args(%4 = %2, %5 = %3) {
      %in_0 = tensor.extract %x[i, j]
      %in_1 = tensor.extract %y[j, k]
      %bb0 = arith.subf %in_0, %in_1 : f64
      %bb1 = arith.mulf %in_0, %in_1 : f64
      %out_1 = tensor.insert %bb0 into %x[i, j]
      %out_0 = tensor.insert %bb1 into %y[j, k]
      scf.yield %out_1, %out_0
    }
    scf.yield %res_1#0, %res_2#1
  }
  scf.yield %res_0#0, %res_0#1
}
```

Conversely, the following is not a well-behaved loop nest due to the highlighted reasons:

```
%res = scf.for i from 0 to M iter_args(%0 = ..., %1 = ...) {
  %res_0:2 = scf.for %j = 0 to N iter_args(%2 = %0, %3 = %1) {
    // Use of tensor computation in the innermost loop
    %in_0 = tensor.extract_slice %x[i] : tensor<?x?xf64> to tensor<?
   xf64>
    %bb0 = arith.mulf %in_0, %in_0 : tensor<?xf64>
    scf.yield %bb0, %bb0
  }
  // More than one loop as a direct child
  scf.for %j = 0 to N iter_args(%2 = %0) {}
  // A value other than %res_0 was yielded
  scf.yield %y
  // This would also result in the nest not being well-behaved,
  // as the order of yielded operands does not match the order
  // they are produced.
  // scf.yield %res_0#1, %res_0#0
}
```

All loop nests generated from lowering `linalg` ops are considered well-behaved loop nests. Additionally, Listing 4.5 is recognized as a valid loop nest despite not being able to be expressed as a `linalg` op.

Once input and output tensors are obtained, their corresponding indexing maps are inferred from their extract and insert ops. In particular, extract/insert ops that index using loop induction variables result in indexing maps that use dimension expressions, while other values produce a special *unknown* value.

Given the adjoint pseudo-code from before:

```
func adj_reduce_some(x, idxs, dresult):
  dx = zeros_like(x)
  for i from 3 to 0:
    dx[idxs[i]] = dresult[i]
```

This results in the following loop nest, expressed as a pseudo `linalg` op:

```
linalg.generic
  { indexing_maps = [(d0) -> (d0), (d0) -> (<unknown>)] }
  ins(dresult)
  outs(dx) {
    ^bb0(%in, %out):
      linalg.yield %in
  }
```

The end result is the pseudo `linalg` op, which is a step towards being able to apply sparse propagation. However, the existing sparse propagation procedure will not work due to the unknown indexing map of the output. To remedy this, observe that the loop nest satisfies a few criteria that allow it to be more predictable. 1) It has a single input operand with an identity indexing map, meaning the loop nest will iterate over every element of its input. 2) Zero values from the input will also be propagated to its output. 3) If its input has only a single nonzero element, then the output must also have a single nonzero element. Thus, our sparse propagation analysis can recognize this case and correctly mark our output as one-hot sparse (sparse along all dimensions), even without statically knowing what position the nonzero value is in. To further propagate sparsity, the compiler-internal `indices` dictionary is updated with the indices of the `tensor.extract` op used to write to the outputs of the

loop nest.

Loop nest analysis is a way to recognize these special cases even when static information (such as the indexing map of the output) is not available at compile time. The information derived from loop nests with missing information is less comprehensive than information derived from `linalg` ops, but this enables sparse propagation to continue to propagate downstream when information is missing or incomplete.

4.4 Summary

This chapter has discussed the novel optimizations of active sparsity and adjoint sparsity, optimizations that take place after automatic differentiation and exploit the high-level information available in the IR. It covers the concepts, analyses, and code generation procedures required to implement both of these optimizations. Following this, we will now evaluate the efficacy of the optimizations covered in the previous two chapters.

Chapter 5

Evaluation

The performance of both forward mode and reverse mode of LAGrad are evaluated in separate stages. As the majority of optimizations in this work are specific to reverse mode, most of the evaluation focuses on that mode. First, individual optimizations are selectively enabled to examine the effect of each optimization's individual contribution. Then, the variant of LAGrad with all optimizations enabled is evaluated against both Enzyme [11] and PyTorch [16]. Finally, the chapter concludes with a brief exploration of forward mode performance compared to its reverse mode counterpart.

Enzyme is chosen as a baseline because it, like LAGrad, performs source-to-source AD in a compiler infrastructure. Enzyme performs differentiation on low-level LLVM IR while LAGrad performs differentiation on high-level tensor MLIR, so the purpose of this comparison is to evaluate the difference made by the level of abstraction of the input programs. Conversely, PyTorch is compared against to examine the difference between performing AD at run time versus compile time while both systems differentiate high-level programs. These two comparison axes are visualized in Figure 5.1.

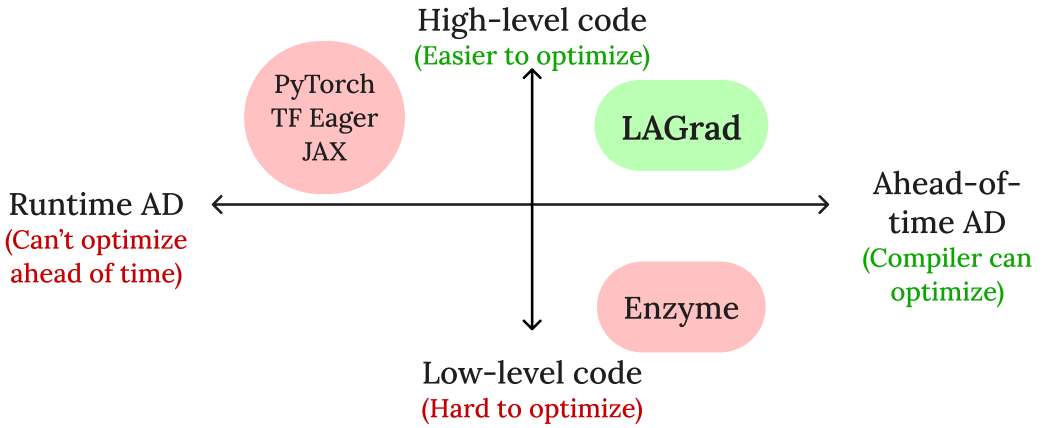

Figure 5.1: The unique position of LAGrad in the automatic differentiation landscape.

5.1 Experimental Methodology

The performance of all AD systems are evaluated using ADBench [9], 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 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 and are thus excluded, while runs longer than 40 minutes are considered timeouts.

In addition, a triangular matrix vector multiplication pullback (TRMV-Row) 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 triangular matrix vector multiplication is evaluated using a single pullback because this is how the computation would commonly appear within the context of a larger reverse-mode computation. The evaluated sizes are [1024, 2048, 4096] for the TRMV-Row 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 via the `task_info` kernel function on Darwin. 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 produce differentiated code. 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 Effect of Optimizations

The optimizations presented in this work are evaluated by enabling them one at a time to augment the baseline pipeline. The benchmarks are evaluated after enabling each optimization in the given order: in-place bufferization, stack buffer promotion, active

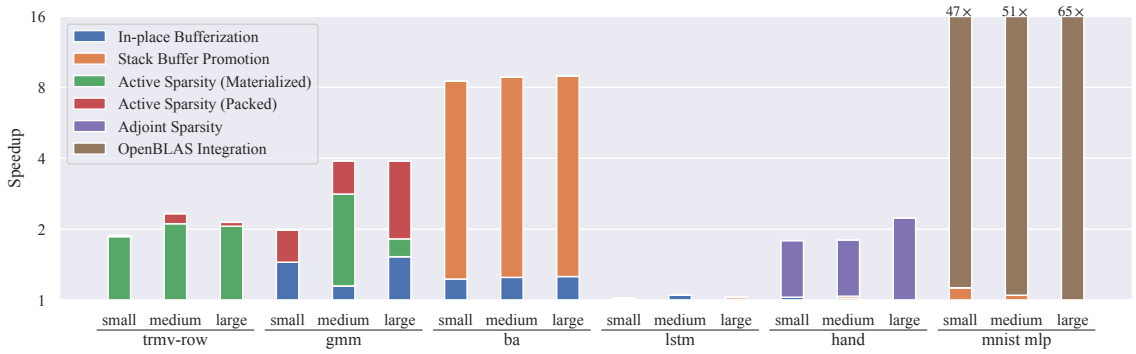

Figure 5.2: Performance impacts of individual optimizations in the LAGrad pipeline. The baseline used is LAGrad run through -O3 without any of the custom optimizations implemented in this work.

sparsity, adjoint sparsity, and library call (OpenBLAS) integration. The cumulative effects of enabling each optimization are summarized in Figure 5.2.

Active Sparsity is evaluated in two stages via first enabling only triangular computation while maintaining full materialization of actively sparse triangular tensors (*Active Sparsity (Materialized)*), then enabling triangular packing to compress the memory representation of tensors (*Active Sparsity (Packed)*). The benchmarks that present opportunities for active sparsity are the TRMV-Row and GMM benchmarks. Note that the GMM benchmark contains a triangular matrix vector multiplication in addition to computation of matrix norms of triangular tensors. The TRMV-Row 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 greater for larger

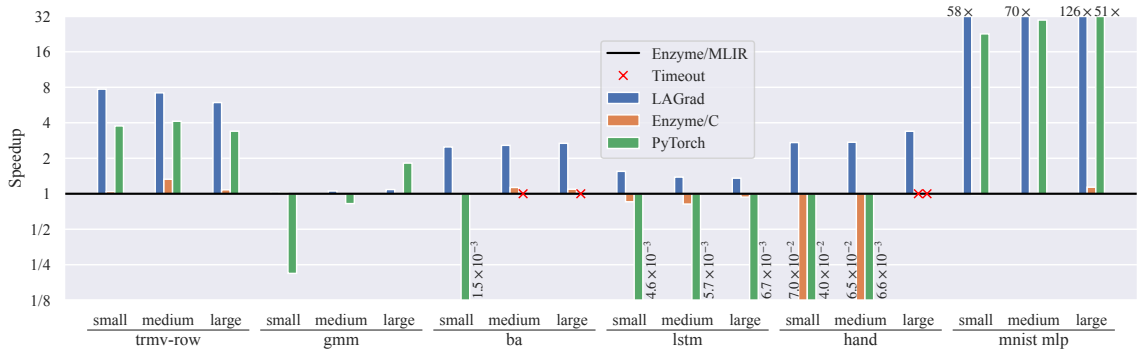

Figure 5.3: Speedup of the fully optimized LAGrad variant vs Enzyme and PyTorch. The baseline is Enzyme performing AD on the translated benchmarks in high-level MLIR, while Enzyme performing AD on the original C programs from ADBench is also included. A red × indicates that a benchmark did not complete within the allotted time limit.

datasets which have Jacobians with a larger 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 computations 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 5.2.

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 [28] routines.

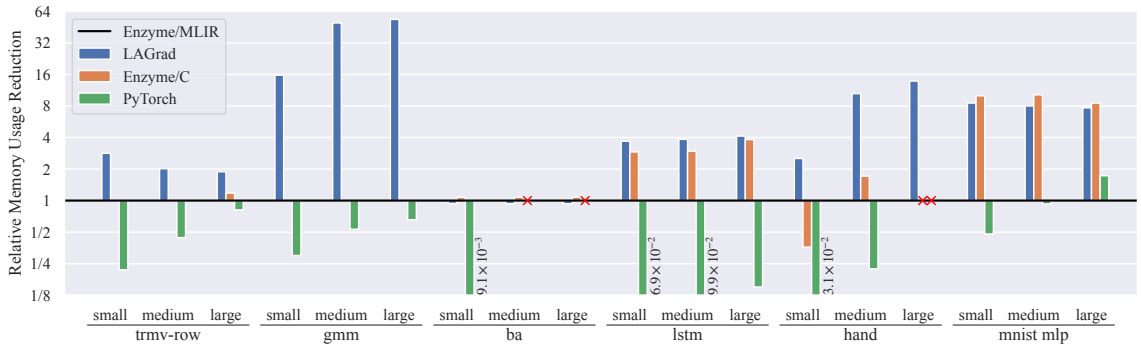

Figure 5.4: Relative peak memory use reduction of LAGrad vs Enzyme and PyTorch (higher is better).

5.3 Comparison with State of the Art

After evaluating individual optimizations, we now turn our attention to comparing the fully optimized LAGrad variant with Enzyme [11]. Results are summarized in Figure 5.3 with geometric means displayed in Table 5.1.

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 by Moses and Churavy [11]. 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 by Moses and Churavy [11].

TRMV-Row demonstrates one of the benefits of performing AD on `linalg` ops. Enzyme differentiates TRMV-Row 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 $\mathcal{O}(n^2)$ 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 triangular tensors, but the burden of programming with the packed representation is placed entirely on the programmer in the case of Enzyme (see Version 3 of Listing 4.3). The primal must directly contain these index computations for Enzyme to generate efficient code, while the LAGrad compiler can automatically generate the same code from an annotated `linalg` op in MLIR.

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 high-performance BLAS 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.

Run Time Variance. LAGrad demonstrates the lowest variance of the measured tools with a relative standard deviation of under 3.7% in its run time on all benchmarks but the multi-layer perceptron, which has a slightly higher relative standard deviation of 7%.

Enzyme shows a similar low variance with a relative standard deviation of under 3.8% on all benchmarks with the exception of the triangular matrix-vector multiplication. On TRMV-Row, Enzyme shows a relative standard deviation of up to 7.2%.

PyTorch has the highest variance in its run time performance among the measured tools. On the ADBench benchmarks it has a relative standard deviation of under 4.1%, though on the multi-layer perceptron its relative standard deviation goes up to 10.6% and on the TRMV-Row benchmark it has a relative standard deviation of up to 34%.

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-Row	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**

Table 5.1: Geometric mean speedups and relative memory reduction of each benchmark across all evaluated datasets.

5.4 Forward Mode

In addition to the focus on reverse mode AD in this work, LAGrad implements forward mode AD. However, the majority of optimizations in this work are reverse mode-specific and do not benefit forward mode AD for the following reasons.

Tape Size Reduction. The gradient tape is fundamentally a structure of reverse mode AD. As tangent numbers are computed in tandem with primal numbers with the same control flow, there are no intermediate values that are overwritten prior to their value being used in the gradient computation. As such, forward mode has no tape to reduce.

Adjoint Sparsity. The applicability of adjoint sparsity in forward mode is complex. On one hand, the computation of full Jacobian matrices in forward mode requires the use of one-hot seed vectors just as in reverse mode, meaning there are similarly many sparse differential values in forward mode. However, there exists less of a clean separation between primal and tangent ops due to the coupled control flow between the two.

For instance, when differentiating a `linalg` op with reverse mode, the result is a pullback

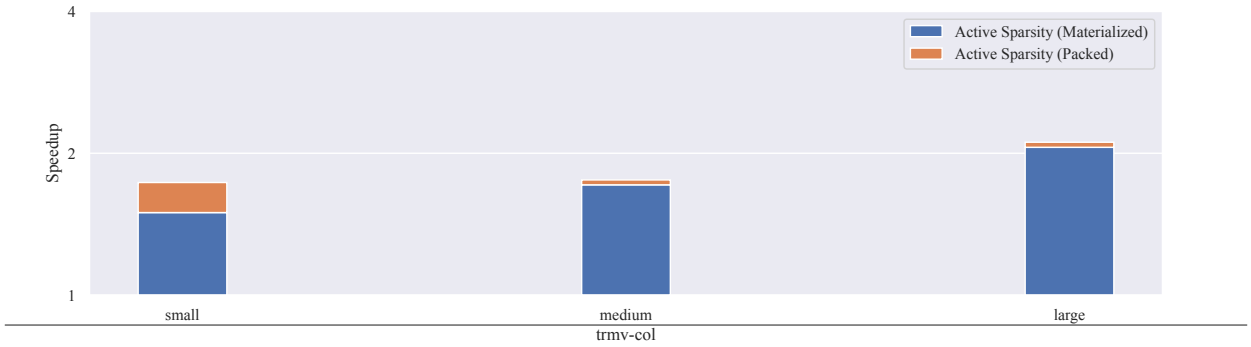

Figure 5.5: TRMV-Col, a forward-mode version of the TRMV-Row benchmark. This demonstrates the speedup of active sparsity in forward mode AD.

which is a distinct op from the primal. In contrast, the equivalent pushforward *augments* the primal op such that both primal and tangent results are computed at the same time. Adjoint sparsity optimizes by skipping over zero values in the tangent values, but applying these same optimizations to forward mode would result in skipping over nonzero values in the primal, leading to incorrect results.

There exists the potential to separate out primal and tangent computation in forward mode, though the resulting complexity of this direction is outside the scope of this work.

Active Sparsity. In contrast to the other optimizations in this work, active sparsity is trivially applicable to forward mode in addition to reverse mode. This is because the notion of propagating differential values through only a subset of array elements is not specific to forward or reverse mode. A corollary of this is that activity analysis (subsection 3.3.1) is completely agnostic to the mode of differentiation. In addition, the code transformations around sparsely active tensors can be run on the augmented pushforward ops because primal and tangent values have the same active sparsity patterns, unlike the sparsity patterns of adjoint sparsity.

Figure 5.5 demonstrates this, showing a modified version of the TRMV-Row benchmark

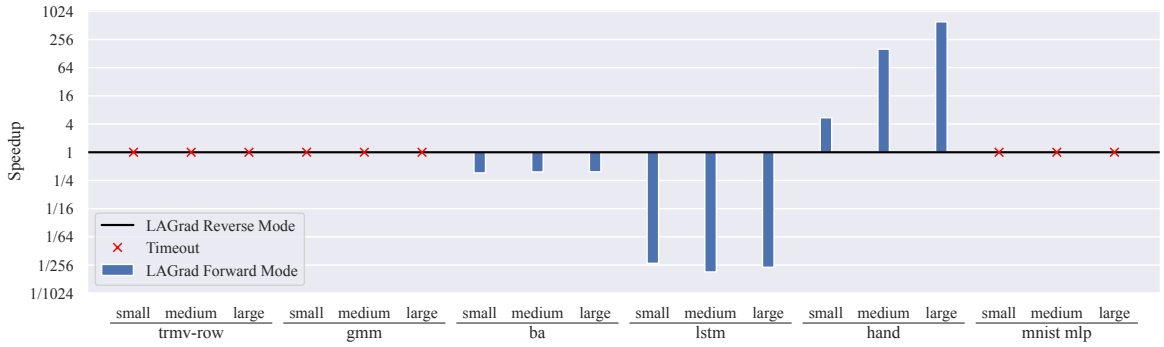

Figure 5.6: Speedup of forward mode LAGrad vs reverse mode.

that computes a single pushforward instead of a single pullback (TRMV-Col). This modified TRMV-Col benchmark demonstrates similar performance benefits from the active sparsity optimization as the reverse mode TRMV-Row benchmark.

5.4.1 Experimental Methodology

The evaluation of forward mode AD in LAGrad is measured in comparison to its reverse mode implementation with all optimizations enabled. Though both Enzyme and PyTorch offer experimental forward mode implementations, neither is able to differentiate the benchmarks evaluated in this work. For that reason, the performance of LAGrad's forward mode is solely compared against its reverse mode implementation with all optimizations enabled.

The same machine, datasets, and experimental methodology is otherwise applied. This includes taking the median run time of 5 runs with 1 warmup run and the timeout limit of 40 minutes.

	TRMV-Row	GMM	BA	LSTM	Hand	MLP
# Inputs	2389	1808276	7.5	415	1	934239
# Outputs	1	1	1	1	3973	1

Table 5.2: Average ratio of inputs to outputs for each benchmark.

5.4.2 Comparison with Reverse Mode

The majority of benchmarks evaluated in this work favour reverse mode due to having significantly more active inputs than active outputs. This is shown in Table 5.2, which lists the average ratio of number of active inputs to active outputs for each benchmark evaluated. This is equivalent to the ratio of number of columns to number of rows of the corresponding Jacobian matrices, and explains the relative number of forward sweeps required to compute the same number of entries computed by one reverse sweep.

Notably, Hand Tracking is the one benchmark where the number of outputs is greater than the number of inputs. This is reflected in how the performance of its forward mode Jacobian computation is several orders of magnitude faster than its reverse mode computation. The relative performance of the other benchmarks is roughly correlated to the relative input-to-output ratios. Results are summarized in Figure 5.6.

5.5 Summary

This chapter has evaluated the effectiveness of performing compile-time automatic differentiation on a high-level language. The comparison against Enzyme has demonstrated the benefit of operating on high-level programs instead of low-level programs in a compile-time system, while the comparison against PyTorch has shown the advantage of

performing AD at compile time instead of run time.

We have also seen the effects of individual optimizations in isolation. While each benchmark shown in Figure 5.2 tends to benefit from a different set of optimizations, all of these optimizations are performed statically and benefit from the high level of abstraction upon which LAGrad operates.

Finally, the exploration of forward mode differentiation has shown that the sparsity optimizations introduced in this thesis have the potential to generalize to forward mode in addition to reverse mode. The deeper exploration of this potential is a promising area for future work.

Chapter 6

Conclusion

As the algorithmic engine that drives the ever-increasing computational needs of deep learning, Automatic Differentiation and its resource cost are significant factors in the computational needs of the modern world. The optimizations presented in this thesis demonstrate that there remain many avenues for optimizing these computational needs. Applications such as masking in transformers [29] and padding in convolutional neural networks [30] both introduce sources of active sparsity, and our evaluation has demonstrated that industry standard AD systems have many cases in which they fail to perform well.

Beyond deep learning, the field of differentiable programming represents a future path of augmenting gradient-based optimization with domain-specific expertise. This has the potential of curbing the trend of merely increasing data and computation. Instead, this direction could actually decrease model size in favour of producing trainable models that incorporate human knowledge.

By targeting the compiler *lingua franca* of MLIR, LAGrad can combine the flexibility of

differentiable programming approaches with the performance of novel optimizations. 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.

6.1 Summary of Contributions

This thesis has demonstrated the benefit of performing source-to-source AD on a high level language. It extends TBR analysis [7] to support high level linear algebra operations and shows how the nature of the input language results in simpler analyses than what other source-to-source AD systems require. It introduces three novel static optimizations that leverage the unique position in the automatic differentiation ecosystem that LAGrad occupies.

These optimizations would be either impossible or more challenging to apply in other systems. Tape size reduction explicitly relies on structured control flow in MLIR, which is a unique characteristic not present in the input languages of other AD systems. Operator-overloading systems fully unroll loops, making tape size reduction infeasible because loop structure is lost. Other compile-time systems operate on SSA form IRs that lack structured control flow constructs. LAGrad is able to automatically discover opportunities to elide the gradient tape, long considered a fundamental challenge of reverse-mode AD.

The two sparsity-based optimizations rely on high level information that can be read from `linalg` ops. Implementing these in low-level intermediate representations would require rediscovering this information, a complex and error-prone task.

6.2 Critical Analysis

Though this thesis demonstrates the performance benefits of the presented approach, it is not without limitations. We now discuss the most pertinent ones.

Manual translation. An important limitation performed in this work is the overhead of translating existing code to MLIR by hand. The textual format of MLIR is verbose due to the explicit information in the IR, which gives the process of manual translation a high engineering overhead. MLIR as a project is relatively new, meaning there is currently a lack of mature front-ends that target it. However, there is ongoing work with examples such as SharkPy and Intel's Python front-end that allow generating MLIR from high-level Python code, and Polygeist which generates MLIR from C and C++. This is a key reason that the number of benchmarks in the evaluation is relatively limited. The ongoing growth of the MLIR front-end ecosystem will remove the need for manual translation in the future.

Expressiveness of the linalg dialect. Another limitation is that the primary performance optimizations in this work depend on the semantic information at the `linalg` dialect level. As discussed in subsection 4.3.2, the `linalg` dialect contains restrictions such that some computation cannot be expressed using its operations. In this case, the potential for applying analyses such as sparse propagation and active sparsity will be limited, meaning the behaviour and performance of LAGrad will be roughly analogous to Enzyme due to the shared usage of the LLVM infrastructure.

Integration with core MLIR optimizations. The majority of optimizations with the greatest impact involved custom analyses and transformations that were implemented with this work. A stated goal of the MLIR infrastructure is to create an ecosystem of general, *composable* transformations that can benefit a wide variety of applications. One such instance is the sparse compiler infrastructure in core MLIR [26], which provides code

generation to optimize both computation and memory of sparse tensors. In contrast, the adjoint sparsity implementation in this work optimizes computation but not memory of sparse tensors, leaving zeros fully materialized in memory. This decision to use a custom lowering instead of the core MLIR sparse infrastructure is made due to the relative immaturity of the MLIR ecosystem; at time of writing, the core sparse infrastructure does not support the full set of ops used by the benchmarks in this work. In the future, improved integration with more mature transformations will further benefit LAGrad.

6.3 Future Work

The field of Automatic Differentiation remains populated with directions for future research.

Beyond Forward and Reverse AD. From an algorithmic perspective, forward- and reverse-mode AD are two extreme ways to traverse the chain rule to compose derivative expressions. One recent extension beyond this is *mixed-mode* automatic differentiation [31], which embeds a piece of forward-mode computation within a larger reverse-mode process. This approach is shown to be general in the specialized case of broadcast operations in GPU kernels, but it remains an open problem to determine if it provides advantages in more general cases.

Jacobian Sparsity and Graph Colouring. The sparsity used in this work are merely two cases where sparsity inherent in Jacobian matrices can be used to optimize their computation. Recall that computation of Jacobian matrices using either forward or reverse mode AD require repeated calls with varying one-hot seed vectors to produce the columns or rows of the Jacobian matrix. If a seed vector is used that has two or more 1s, the result of the AD process will be the *sum* of the respective columns/rows. A corollary of this is

that if two or more columns of the Jacobian are sparse such that their nonzero values are never in the same row, running AD with both columns seeded will compute the nonzero elements of both columns in a single pass. Then, both columns are reconstructed from the known sparsity pattern. This approach can greatly reduce the number of passes required to compute Jacobians, and is shown to be equivalent to a graph coloring problem [32, 33].

Existing approaches assume a known sparsity pattern, but fine-grained activity analysis between individual elements of input and output vectors has the potential to automatically determine the sparsity patterns of Jacobians. A challenge of this approach is that such a fine-grained analysis scales with the number of elements in an array and requires reasoning about data flow in a much more complex fashion than coarse-grained analyses at the tensor level.

Beyond Tensors and Structured Control Flow. While restricting the input language to tensor MLIR with structured control flow is shown to simplify the necessary optimizations, there exist applications that are most naturally expressed using code with memory side effects and require unstructured control flow. The extensibility of MLIR imply that users will continue to create new, custom dialects that match their specific use cases. A future direction made feasible by MLIR is to define the minimum set of information required to differentiate arbitrary ops in arbitrary dialects. Extending the optimizations presented in this work to a greater diversity of dialects, computation models, and hardware accelerators remains an open problem.

Bibliography

[1] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J. Zico Kolter. End-to-end differentiable physics for learning and control. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[2] Li Li, Stephan Hoyer, Ryan Pederson, Ruoxi Sun, Ekin D. Cubuk, Patrick Riley, and Kieron Burke. Kohn-sham equations as regularizer: Building prior knowledge into machine-learned physics. *Phys. Rev. Lett.*, 126:036401, Jan 2021.

[3] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[4] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), jul 2018.

[5] Mike Innes, Alan Edelman, Keno Fischer, Christopher Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. A differentiable programming system to bridge machine

learning and scientific computing. *CoRR*, abs/1907.07587, 2019.

[6] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[7] Laurent Hascoët, Uwe Naumann, and Valérie Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8):1401–1417, 2005.

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

[9] Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33:1–14, 02 2018.

[10] Michael J Innes. Don't unroll adjoint: Differentiating ssa-form programs. In *Workshop on Systems for ML*, 2018.

[11] William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020.

[12] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[13] Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[14] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2), mar 2008.

[15] Bart van Merrienboer, Olivier Breuleux, Arnaud Bergeron, and Pascal Lamblin. Automatic differentiation in ml: Where we are and where we should be going. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[16] 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. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

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

[18] 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. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[20] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, USA, 1979. AAI8007856.

[21] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 109–121, New York, NY, USA, 1986. Association for Computing Machinery.

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

[23] Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3), 2013.

[24] Bart van Merrienboer, Dan Moldovan, and Alexander Wiltschko. Tangent: Automatic differentiation using source-code transformation for dynamically typed array

programming. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, oct 1991.

[26] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022.

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

[28] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, 2012.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q.

Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[31] Jarrett Revels, Tim Besard, Valentin Churavy, Bjorn De Sutter, and Juan Pablo Vielma. Dynamic automatic differentiation of gpu broadcast kernels. In *Workshop on Systems for ML*, 2018.

[32] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM Rev.*, 47:629–705, 2005.

[33] Jan Hückelheim, Michel Schanen, Sri Hari Krishna Narayanan, and Paul Hovland. Vector forward mode automatic differentiation on simd/simt architectures. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.