# Compiler Optimizations for Higher-Order Automatic Differentiation

Vimarsh Sathia
vsathia2@illinois.edu
University of Illinois Urbana Champaign
Urbana, Illinois, USA

Siyuan Brant Qian
siyuanq4@illinois.edu
University of Illinois Urbana Champaign
Urbana, Illinois, USA

Jan Hückelheim
jhueckelheim@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Paul Hovland
hovland@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

William S. Moses
wsmoses@illinois.edu
University of Illinois Urbana Champaign
Urbana, Illinois, USA

## Abstract

Higher-order derivatives have critical applications in scientific computing and machine learning, ranging from neural ODEs to Hessian-based optimizers. Although existing approaches to compute these derivatives use automatic differentiation (AD), the generated code to compute higher-order derivatives is usually suboptimal, leaving room for optimizations on the table.

We show how compiler optimizations can improve the performance of higher-order AD in tensor programs. Our key insight is that AD-generated derivative code often contains exploitable high-level structure, such as common sub-expressions and symmetric mixed derivatives. To leverage these structures, we implement compiler passes in the MLIR StableHLO intermediate representation, including constant propagation and other tensor-specific rewrites. Preliminary benchmarks on second-order Laplacian computations in deep neural networks show speedups of upto 1.3×.

These findings suggest that compiler-level optimizations offer a promising path toward making higher-order AD more practical and efficient, particularly for large-scale applications that rely on complex derivative computations.

## Keywords

automatic differentiation, compiler optimizations, tensor programs, scientific computing

**ACM Reference Format:**
Vimarsh Sathia, Siyuan Brant Qian, Jan Hückelheim, Paul Hovland, and William S. Moses. 2025. Compiler Optimizations for Higher-Order Automatic Differentiation. In *Proceedings of Workshop on Differentiable Parallel Programming (PPoPP '25).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn


*PPoPP '25, Las Vegas, NV*

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Automatic differentiation (AD) has emerged as a powerful and increasingly popular tool for computing exact derivatives in differential programming. Although most widely used AD frameworks [2, 10] are heavily optimized for first-order derivatives (owing to the dominance of backpropagation in machine learning), second or higher-order derivatives are steadily gaining traction. Examples include Hessian-based ML optimization [3] and neural ordinary differential equations (ODEs) [1].

However, our preliminary findings indicate that simply relying on standard AD frameworks or existing higher-order AD solutions in these applications often leads to suboptimal performance. We argue that well-tuned compiler optimizations and scheduling techniques can bridge this performance gap.

## 2 Current Approaches and Issues

Existing methods for computing higher-order derivatives can broadly be categorized into two approaches:

***Repeated Composition of First-Order Derivatives.*** A straightforward strategy is to repeatedly apply first-order AD operators until the desired order is reached. This approach is appealing because it reuses mature AD frameworks like PyTorch [10], JAX [2], and Enzyme [7–9], all of which support source-to-source transformations. However, there are some caveats:

- **Symmetry in mixed derivatives:** First-order operators cannot reveal exploit structure in higher-order derivatives, such as the symmetry of partial derivatives. Consequently, naively expanding repeated derivatives can lead to an exponential blow-up in terms. For example, for a function $f : \mathbb{R}^n \to \mathbb{R}$, computing all second-order partial derivatives $\forall i, j \in \{1, 2, 3 \cdots, n\}$, $\frac{\partial^2 f}{\partial x_i \partial x_j}$ results in $n^2$ evaluations, even though only $\frac{n(n+1)}{2}$ of them are unique due to symmetry.
- **Sparse computations:** In many practical scenarios, only a small subset of mixed derivatives are actually needed for subsequent computations. For instance, if a downstream computation only needs the principal diagonal of the Hessian, we can skip computing all partial derivatives and focus

**Table 1: Time for different methods to compute the Laplacian of a 64 layer MLP. All evaluations are performed on a CPU build of `jax`. Reported time is averaged over 10 evaluations, and is in seconds. The minimum time is highlighted for every row.**

| hidden layer size | *Our optimizations* | `trace(jax.hessian)` | `jax.jvp(jax.grad)` | `jax.jet` |
|---|---|---|---|---|
| 256 | disabled | 3.687 | 3.619 | **2.364** |
| | enabled | 3.196 | 3.104 | **1.872** |
| 512 | disabled | 13.823 | 14.031 | **12.580** |
| | enabled | 13.743 | 13.894 | **9.759** |
| 384 | disabled | 7.155 | 7.507 | **6.280** |
| | enabled | 4.217 | 4.559 | **3.049** |
| 128 | disabled | **0.761** | 0.763 | 0.827 |
| | enabled | 0.771 | **0.765** | 0.901 |
| | Geomean speedup | 1.181 | 1.179 | **1.324** |

solely on $\frac{\partial^2 f}{\partial x_i^2}$, and map this to a sparse tensor operation, saving on storage and compute.

***Taylor AD.*** An alternative strategy is to compute higher-order derivatives via a Taylor series expansion, as described in Chapter 13 of Griewank and Walther [4]. This method is implemented for forward-mode AD in the `jax.jet` module [1], which inherently handles symmetries in mixed partials and thus mitigates some of the exponential blow-up issues noted in repeated composition.

However, all Taylor AD implementations rely on **operator overloading**, resulting in large operator call overheads at runtime. For example, in `jax.jet`, we observed that transcendental operators like `lgamma` were not folded into constants by XLA during compile time.

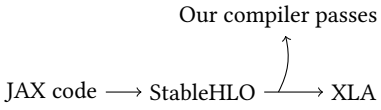


**Figure 1: Compilation pipeline showing where our compiler passes are inserted, before dispatching code to XLA[11]**

## 3 Leveraging Compiler Optimizations

We ask ourselves the following research question:

*How do we effectively optimize code which requires the use of higher-order derivatives?*

We believe that the answer to this question lies in designing or adapting compiler techniques can exploit the high-level structure exposed by derivative operators. Specifically:

***Constant Propagation.*** Constant Propagation can help reduce runtime overheads in operator overloading. Especially in the case of `lgamma` (represented in StableHLO as `chlo.lgamma`), we have the following transformation at compile time:

```
// Before constant propogation
%cst = stablehlo.constant dense<1.00e+01> : tensor<f32>
%0 = chlo.lgamma %cst : tensor<f32> -> tensor<f32>

// After constant propogation
%0 = stablehlo.constant dense<4.7684E-7> : tensor<f32>
```

***Common Subexpression Elimination(CSE).*** CSE, after augmenting it with the information about mixed partial derivatives, can simplify the CFG corresponding to the output derivative.

***Kernel Rewriting.*** We can leverage the sparsity of the derivative computation by propagating them into future operations, and rewrite them into more storage or compute efficient operators. For example, since we know that the Hessian output $H$ is symmetric, we can replace all $gemm(H, B)$ calls with $symm(H, B)$ where $symm$ is a matmul routine specialized for symmetric matrices. This can also be done for sparse computations.

***Tensor Rewrites.*** Tensor rewrites have emerged as a very important method to optimize tensor programs.[5]. These rewrites are particularly effective for higher-order derivatives where complex mathematical expressions and repeated tensor operations are common.

## 4 Implementation in MLIR and JAX

We have a prototype implementation in JAX and MLIR[6]. Figure 1 describes the compilation flow we use to optimize a general jax program. Currently, we run all our optimization passes after generating the derivative code.

For our preliminary results, we compute the Laplacian of a Multi Layer Perceptron defined in `jax`, and compare runtimes on CPU. The results are shown for various hidden layer widths, and 3 different ways of computation in Table 1.

## 5 Conclusions and Future Work

Our preliminary results suggest that compiler optimizations can significantly improve higher-order AD performance in tensor programs. The observed speed-ups indicate that a lot of performance is left on the table. In the future, we plan to add support for the rest of the compiler passes like Kernel Rewriting and CSE. In order to effectively add support for the same, we need to propagate attribute information across the derivative boundary.

## References


[1] Jesse Bettencourt, Matthew J. Johnson, and David Duvenaud. 2019. Taylor-Mode Automatic Differentiation for Higher-Order Derivatives in JAX. In *Program Transformations for ML Workshop at NeurIPS 2019*. https://openreview.net/forum?id=SkxEF3FNPH

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/jax-ml/jax

[3] Mathieu Dagréou, Pierre Ablin, Samuel Vaiter, and Thomas Moreau. 2024. How to compute Hessian-vector products?. In *The Third Blogpost Track at ICLR 2024*. https://openreview.net/forum?id=rTgjQtGP3O

[4] Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives* (second ed.). Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898717761 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717761

[5] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

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

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

[8] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. doi:10.1145/3458817.3476165

[9] William S. Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. 2022. Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 60, 18 pages.

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

[11] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.