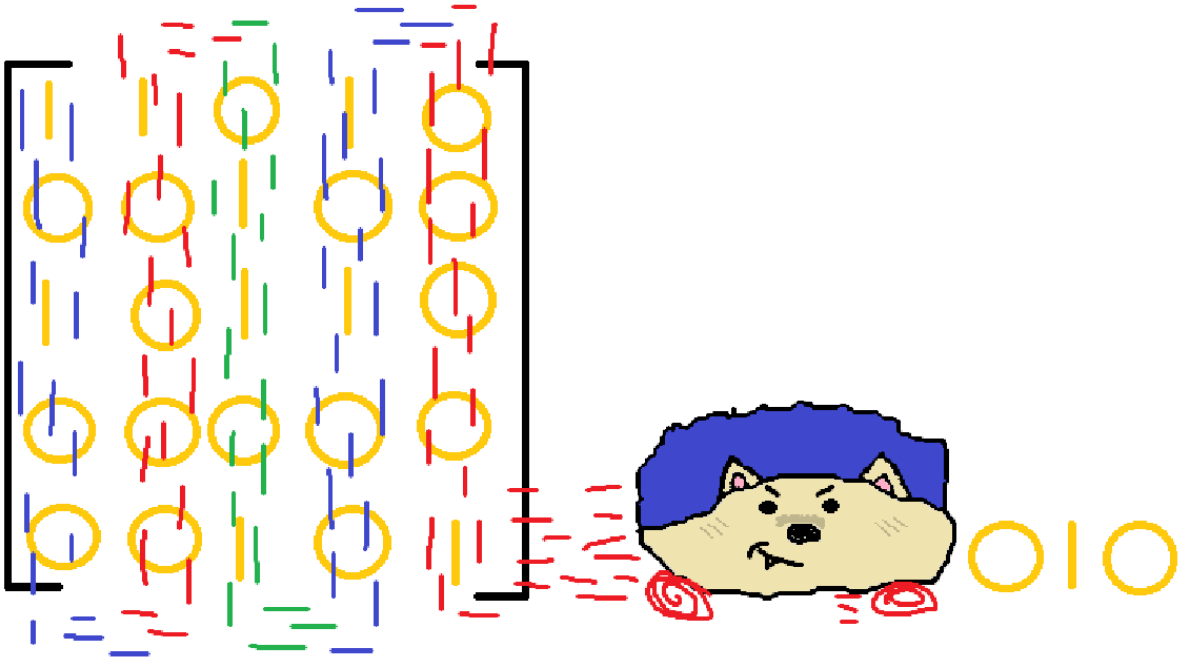
UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE



# BSc Thesis

Elias Fontain Smedegaard

# Sparse Jacobian Computation in Futhark

Exploiting Known Sparsity Patterns Using Graph Coloring

Advisor: Troels Henriksen

This thesis has been submitted to the Department of Computer Science,
University of Copenhagen on June 9, 2025.

## Abstract

Automatic differentiation makes it possible to compute derivatives of functions implemented as programs, but computing a full dense Jacobian can be wasteful when the Jacobian is sparse. This thesis investigates how a known Jacobian sparsity pattern can be exploited in Futhark to compute only the structurally nonzero entries more efficiently.

The thesis presents a sparse Jacobian library that accepts user-provided sparsity patterns, represents them mainly in compressed sparse row format, and colors the corresponding bipartite graph to construct compressed seed vectors. The implementation supports both a forward-mode pipeline based on Jacobian-vector products and a reverse-mode pipeline based on vector-Jacobian products. Coloring is separated from evaluation, allowing colorings to be reused for repeated computations with the same sparsity pattern.

Correctness tests compare the sparse results with dense Jacobians restricted to the known sparsity pattern. The benchmarks show that sparse compression can provide large speedups when the number of colors is small, especially for the D2 CPU variant, while BGPC performance is more problem- and backend-dependent.

# Contents

# 1 Introduction

Automatic differentiation (AD) makes it possible to compute derivatives of functions implemented as programs. For functions with many inputs and outputs, however, the full Jacobian can be large. This is often wasteful, because many Jacobians are sparse, meaning that a large fraction of their entries are structurally zero and therefore known in advance not to contribute to the result. This thesis investigates how such known sparsity can be exploited in Futhark. The central problem is: given a differentiable Futhark function and a user-provided Jacobian sparsity pattern, can we compute the structurally nonzero Jacobian entries more efficiently than by computing the full dense Jacobian?

To answer this question, we design and implement a Futhark library for sparse Jacobian computation. The library accepts a user-provided sparsity pattern, either as a dense Boolean pattern or in compressed sparse row (CSR) format. The main pipeline uses CSR as the sparse representation, colors the pattern to find structurally non-conflicting rows or columns, and uses the resulting colors to construct compressed seed vectors. The implementation supports both a forward-mode pipeline based on Jacobian-vector products (JVPs) and a reverse-mode pipeline based on vector-Jacobian products (VJPs). Coloring is separated from the compressed AD evaluation, so that colorings can be reused whenever the same sparsity pattern is used for several input points.

The graph-coloring approach used in this thesis is based on the approach to sparse Jacobian computation in *What Color Is Your Jacobian?* by Gebremedhin, Manne, and Pothen [4]. The implemented coloring routines are based on greedy coloring ideas from this work and on optimistic bipartite graph partial coloring techniques from Tas, Kaya, and Saule [10]. The contribution of this thesis is not a new graph-coloring theory or a new AD method, but the design and implementation of these ideas in Futhark, including compressed forward- and reverse-mode sparse Jacobian pipelines, reusable colorings, correctness tests, and performance benchmarks. The implementation, tests, and benchmark code are publicly available on GitHub.[1]

The thesis is organized as follows. Section 2 introduces the background on Jacobians, automatic differentiation, sparse Jacobian compression, graph coloring, and Futhark. Section 3 describes the design and implementation of the sparse Jacobian library. Section 4 evaluates correctness and performance. Section 5 presents the conclusion of the thesis.

[1] `https://github.com/Nyfo/sparse`

# 2 Background

## 2.1 Jacobians and sparsity patterns

We consider a differentiable function

$$f : \mathbb{R}^n \to \mathbb{R}^m. \tag{1}$$

Here, $n$ is the number of differentiable input components, and $m$ is the number of output components. This notation also covers Futhark functions whose differentiable inputs and outputs are arrays of floats, since such arrays can be flattened and treated as vectors when differentiating.

We let $x \in \mathbb{R}^n$. With zero-based indexing, the Jacobian of $f$ at $x$ is the matrix

$$J_f(x) = \begin{bmatrix} \frac{\partial f_0}{\partial x_0}(x) & \cdots & \frac{\partial f_0}{\partial x_{n-1}}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{m-1}}{\partial x_0}(x) & \cdots & \frac{\partial f_{m-1}}{\partial x_{n-1}}(x) \end{bmatrix} \in \mathbb{R}^{m \times n}. \tag{2}$$

The entry $J_f(x)_{ij}$ describes how output component $f_i$ changes with respect to input component $x_j$. As such, the Jacobian collects all first-order partial derivatives of $f$ in a single matrix.

In many applications, each output depends only on a small subset of the inputs, so many Jacobian entries are guaranteed to be zero independently of the specific value of $x$ [4]. Such entries are structurally zero, and the Jacobian is sparse when many entries have this property. To describe this structure, we define a sparsity pattern $S \in \{0,1\}^{m \times n}$, where

$$S_{ij} = \begin{cases} 1 & \text{if } J_f(x)_{ij} \text{ may be nonzero,} \\ 0 & \text{if } J_f(x)_{ij} \text{ is structurally zero.} \end{cases} \tag{3}$$

Here, structurally zero means that the derivative is zero because of the dependency structure of the function, not just because of one particular input value.

For example, consider the following function $f : \mathbb{R}^4 \to \mathbb{R}^4$ defined by

$$f(x_0, x_1, x_2, x_3) = \begin{bmatrix} \sin(x_0) + x_2^3 \\ x_1 x_3 \\ x_2^2 + 3x_3 \\ x_0 x_1 \end{bmatrix}. \tag{4}$$

Its Jacobian is:

$$J_f(x) = \begin{bmatrix} \cos(x_0) & 0 & 3x_2^2 & 0 \\ 0 & x_3 & 0 & x_1 \\ 0 & 0 & 2x_2 & 3 \\ x_1 & x_0 & 0 & 0 \end{bmatrix}. \tag{5}$$

The zero entries are structurally zero because their output components do not depend on the corresponding input variables. For example, $f_0$ depends on $x_0$ and $x_2$, but not on $x_1$ or $x_3$. Therefore, the entries $(0,1)$ and $(0,3)$ are structurally zero. In contrast, $3x_2^2$ is zero when $x_2 = 0$, but it is not structurally zero.

It therefore has the following sparsity pattern:

$$S = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}. \tag{6}$$

As such, this pattern records which entries may be nonzero, not which entries happen to be numerically nonzero at some particular input.

In this thesis, we assume that the sparsity pattern is provided by the user. This pattern determines which Jacobian entries the sparse pipelines need to compute.

## 2.2 Automatic Differentiation, JVP, and VJP

AD is a technique for computing derivatives of functions that are implemented as programs. Unlike finite differences, AD does not approximate derivatives by evaluating the function at slightly changed input values. Instead, AD computes derivatives by applying the chain rule to the operations performed by the program [1].

Given a differentiable function $f : \mathbb{R}^n \to \mathbb{R}^m$, Futhark provides AD through the functions `jvp` and `vjp` [2]. In the notation used here, $x \in \mathbb{R}^n$ denotes the point at which the derivative is evaluated. We use $v \in \mathbb{R}^n$ for a forward-mode seed vector and $u \in \mathbb{R}^m$ for a reverse-mode seed vector.

The internal implementation of AD in Futhark is not the focus of this thesis. We use `jvp` and `vjp` as the AD interface and focus on how these functions can be seeded and combined to compute sparse Jacobians efficiently.

Forward mode computes a JVP. In the notation used here, we write

$$\texttt{jvp}\ f\ x\ v = J_f(x)v. \tag{7}$$

This equation describes the result mathematically. The full Jacobian matrix is not explicitly constructed first. Instead, AD computes the product $J_f(x)v$.

If $v = e_j$, where $e_j \in \mathbb{R}^n$ is the elementary vector with entry $j$ equal to 1 and all other entries equal to 0, then

$$J_f(x)e_j = \begin{bmatrix} \frac{\partial f_0}{\partial x_j}(x) \\ \vdots \\ \frac{\partial f_{m-1}}{\partial x_j}(x) \end{bmatrix}. \tag{8}$$

Thus, one JVP with $v = e_j$ recovers column $j$ of the Jacobian. A full dense Jacobian can therefore be reconstructed using $n$ calls to `jvp`, one for each input direction.

Reverse mode computes a VJP. In the notation used here, we write

$$\texttt{vjp}\ f\ x\ u = u^T J_f(x). \tag{9}$$

As for JVPs, this product is computed without explicitly constructing the full Jacobian first.

If $u = e_i$, where $e_i \in \mathbb{R}^m$ is the elementary vector with entry $i$ equal to 1 and all other entries

equal to 0, then

$$e_i^T J_f(x) = \begin{bmatrix} \frac{\partial f_i}{\partial x_0}(x) & \cdots & \frac{\partial f_i}{\partial x_{n-1}}(x) \end{bmatrix}. \tag{10}$$

Thus, one VJP with $u = e_i$ recovers row $i$ of the Jacobian. A full dense Jacobian can therefore be reconstructed using $m$ calls to `vjp`, one for each output direction.

The functions `jvp` and `vjp` form the basis for the dense and sparse Jacobian methods used in this thesis. The dense baselines use elementary vectors to recover one column or one row at a time. The sparse methods instead use more general seed vectors, where several entries may be nonzero. These seed vectors combine several non-conflicting columns or rows in a single AD call, which is the basis of sparse Jacobian compression [4]. This idea is described in the next subsection.

## 2.3 Sparse Jacobian Compression

The dense methods from the previous subsection use elementary seed vectors to recover one Jacobian column or row at a time. Sparse Jacobian compression still uses JVPs and VJPs, but replaces elementary seed vectors with compressed seed vectors that contain several nonzero entries. The key idea is that several columns can be computed in the same AD call if their structurally nonzero entries do not overlap [4].

First consider column compression using JVP. Suppose that the columns have been divided into groups

$$C_0, \ldots, C_{p-1}, \tag{11}$$

where no two columns in the same group have a structurally nonzero entry in the same row. For each group $C_k$, we use a seed vector $s_k \in \{0,1\}^n$ with ones at the column indices in $C_k$ and zeros everywhere else. A JVP with this seed gives

$$z_k = J_f(x) s_k. \tag{12}$$

Since $s_k$ may contain several ones, $z_k$ is the sum of the Jacobian columns in $C_k$. This sum can still be separated afterwards because each row receives a contribution from at most one column in the group.

We use the Jacobian from Equation 5 as a running example. The columns can for example be grouped as

$$C_0 = \{0, 3\}, \qquad C_1 = \{1, 2\}. \tag{13}$$

The following matrix shows this grouping using colors. The blue entries belong to columns in $C_0$, and the red entries belong to columns in $C_1$:

$$J_f(x) = \begin{bmatrix} \cos(x_0) & 0 & 3x_2^2 & 0 \\ 0 & x_3 & 0 & x_1 \\ 0 & 0 & 2x_2 & 3 \\ x_1 & x_0 & 0 & 0 \end{bmatrix}. \tag{14}$$

The grouping is valid because entries with the same color never occur in the same row. Therefore,

each color can be computed using one compressed seed vector:

$$s_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad s_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{15}$$

The blue seed selects columns in $C_0$, while the red seed selects columns in $C_1$. The two compressed JVPs are

$$z_0 = J_f(x)s_0 = \begin{bmatrix} \cos(x_0) \\ x_1 \\ 3 \\ x_1 \end{bmatrix}, \qquad z_1 = J_f(x)s_1 = \begin{bmatrix} 3x_2^2 \\ x_3 \\ 2x_2 \\ x_0 \end{bmatrix}. \tag{16}$$

Together, these form a compressed Jacobian representation with two columns instead of four:

$$Z = \begin{bmatrix} z_0 & z_1 \end{bmatrix} = \begin{bmatrix} \cos(x_0) & 3x_2^2 \\ x_1 & x_3 \\ 3 & 2x_2 \\ x_1 & x_0 \end{bmatrix}. \tag{17}$$

The sparsity pattern together with the column grouping tells us how to recover the original nonzero entries from this compressed representation. For example, column 3 belongs to $C_0$, so the entry $J_f(x)_{2,3} = 3$ is found in row 2 of $z_0$. Similarly, column 2 belongs to $C_1$, so the entry $J_f(x)_{0,2} = 3x_2^2$ is found in row 0 of $z_1$.

Row compression works in the same way with VJP. Instead of grouping columns, we group rows that do not have structurally nonzero entries in the same column. A reverse-mode seed vector can then select several output components at once, and the resulting VJP contains the compressed row information for that color group [4].

Sparse Jacobian compression therefore reduces the number of AD calls by replacing many elementary seed vectors with fewer compressed seed vectors. The concrete sparse matrix representation used in the implementation is described in Section 3.2. The remaining question is how to find valid groups of non-conflicting columns or rows. This is the graph coloring problem described in the next subsection.

## 2.4 Graph Coloring for Sparse Jacobians

The graph-coloring formulation in this subsection follows the sparse Jacobian coloring approach described in *What Color Is Your Jacobian?* [4].

For column compression, two columns are in conflict if they both have a structurally nonzero entry in the same row. Such columns cannot be placed in the same group, because their contributions would be added into the same entry of the compressed JVP.

A convenient way to represent these conflicts is through a bipartite graph. Given a sparsity pattern $S$, we introduce one row vertex $r_i$ for each row of $S$ and one column vertex $c_j$ for each

column of $S$. An edge connects $r_i$ and $c_j$ exactly when $S_{ij} = 1$, which means the row $r_i$ has an element at column $c_j$. Figure 1 shows this for the running example. The figure also summarizes the connection between the colored Jacobian, the compressed representation, and the bipartite graph.

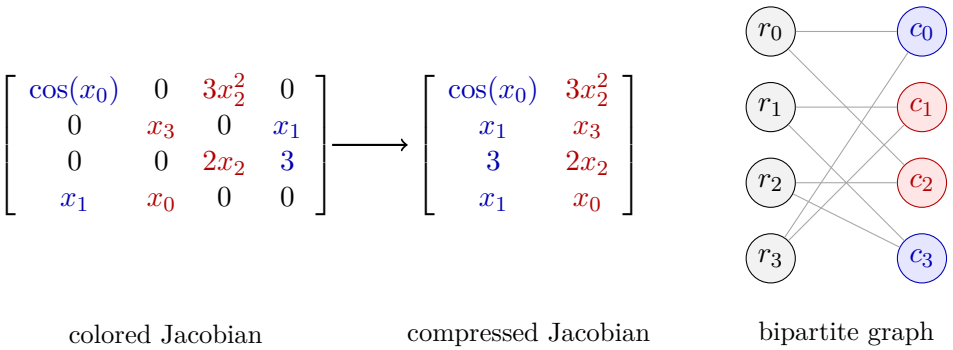



Figure 1: Column compression and graph coloring for the running example. The same coloring is shown as a partial distance-2 coloring of the column vertices in the bipartite graph.

In the bipartite graph, column conflicts appear as shared row neighbors. For example, $c_1$ and $c_3$ are both connected to $r_1$, so they cannot have the same color. Otherwise, row 1 of the compressed JVP would receive contributions from both columns. In contrast, $c_0$ and $c_3$ do not share any row neighbor, so they may be assigned the same color, which is blue in this case.

Finding valid column groups is therefore equivalent to coloring the column vertices such that column vertices with a shared row neighbor receive different colors. Such column vertices are at distance two from each other through a row vertex in their paths. In Figure 1, this gives the two groups

$$C_0 = \{0, 3\}, \qquad C_1 = \{1, 2\}, \tag{18}$$

which are the groups used for the compressed JVPs in Section 2.3.

This is called a partial distance-2 coloring of the bipartite graph [4]. The word partial means that only one side of the bipartite graph is colored. For column compression, the column vertices are colored, while the row vertices only define which columns are in conflict. The coloring is partial, but the Jacobian computation still recovers all structurally nonzero entries.

Row compression works in the same way. For VJP-based compression, the row vertices are colored instead of the column vertices. Two rows are in conflict if they have a structurally nonzero entry in the same column, and each row color gives one compressed VJP seed vector.

Graph coloring therefore provides the step from the sparsity pattern to the compressed AD evaluations. The colors determine the groups $C_k$, the groups determine the compressed seed vectors $s_k$, and the number of colors determines the number of JVP or VJP calls. For the example, this reduces the number of calls from four to two. The concrete greedy coloring algorithms used in the implementation are described in Section 3.4.

## 2.5 Futhark and Backend Considerations

Futhark is a functional, data-parallel array language designed to compile to efficient parallel code [6]. Much of the parallelism in Futhark programs is expressed through operations on whole arrays, including second-order array combinators (SOACs). The compiler translates this array-level parallelism into code for different backends. This makes Futhark a suitable setting for the sparse Jacobian pipeline, where inputs, seed vectors, color assignments, derivative values, and sparse matrix data are represented as arrays.

This thesis uses Futhark's AD functions `jvp` and `vjp` as the interface for derivative computation [2]. The sparse methods do not change how Futhark computes derivatives internally. Instead, they change how `jvp` and `vjp` are seeded, and how their outputs are reconstructed using a known sparsity pattern.

Performance in Futhark depends on the chosen backend. The `c` backend provides sequential CPU execution, while the `multicore` and `cuda` backends use parallel CPU and GPU execution, respectively. Since the same source program can be compiled to different backends, backend choice is important when evaluating whether the sparse Jacobian pipeline benefits from parallel execution.

Sparse Jacobian computation separates work that depends only on the sparsity pattern from work that depends on the numerical input. The first part is the pattern-dependent preparation phase. The main pipeline assumes that the sparsity pattern is already provided in CSR form, so this phase mainly consists of computing a coloring. Its cost depends on the nonzero structure rather than on the particular input point $x$. Even though coloring is only a preparation step, it can still be a significant part of the total sparse pipeline cost [10].

The second part is the point-dependent derivative evaluation. Once the colors have been computed, the implementation constructs one seed vector per color and evaluates one compressed JVP or VJP for each seed.

This separation is useful in the evaluation because coloring and derivative evaluation may behave differently on different backends. The experiments therefore separate coloring from precolored sparse evaluation on the multicore CPU backend and compare selected dense and sparse cases on the multicore CPU and CUDA GPU backends.

# 3 Library Design and Implementation

## 3.1 Design Goal and Assumptions

The previous section introduced the relevant background on sparse Jacobians, AD, sparse Jacobian compression, and graph coloring. This section describes how these ideas are used to design and implement a Futhark library for sparse Jacobian computation.

The main design goal is to exploit a known sparsity pattern to reduce the number of AD passes needed to compute the structurally nonzero entries of a Jacobian. The library assumes that the user provides this sparsity pattern. Discovering sparsity patterns automatically is outside the scope of this project.

The intended scalable pipeline uses a CSR sparsity pattern as input and returns the result in compressed form. The CSR input represents the sparsity pattern using row-wise and column-wise adjacency arrays, which are described in the next subsection. This makes the pattern representation proportional to the number of structurally nonzero entries rather than the full matrix size $mn$. The compressed output is the result closest to the actual AD computation: one derivative result is produced per color. CSR and dense output formats are also provided as reconstructed convenience formats for validation, debugging, and small examples.

The library supports sparse Jacobian computation using both forward-mode and reverse-mode AD. The forward-mode implementation uses JVPs, while the reverse-mode implementation uses VJPs, both through the Futhark AD API. Since the better mode depends on the sparsity pattern, the library also provides an automatic mode that compares the number of colors required by the JVP and VJP approaches and chooses the one requiring fewer AD passes.

The rest of this section describes the sparsity pattern representation, the overall sparse Jacobian pipeline, the coloring algorithms, the JVP and VJP implementations, automatic mode selection, reusable preprocessing, and the supported output formats.

## 3.2 Compressed Sparse Row Representation

A sparsity pattern records which entries of the Jacobian may be nonzero. The library uses this pattern to decide which derivative values must be computed and stored.

The simplest representation is a dense boolean matrix of type `[m][n]bool`. This representation has the same shape as the Jacobian and is easy to inspect, which makes it useful for testing and small examples. However, it requires $O(mn)$ storage even when the number of structurally nonzero entries is much smaller than $mn$.

For larger sparse Jacobians, the library uses compressed sparse row (CSR) arrays, a standard sparse matrix format [7]. In the following, nnz is the number of structurally nonzero entries. In the implementation, a row-wise sparsity pattern is represented by two arrays, `row_offs` of type `[m+1]i64` and `row_idx` of type `[nnz]i64`, where `row_offs` stores where each row begins and `row_idx` stores the column indices of the structurally nonzero entries. The name `row_idx` refers to the row-wise representation, even though the stored indices are column indices. Thus, the nonzero columns in row $i$ are stored in

$$\texttt{row\_idx}[\texttt{row\_offs}[i] : \texttt{row\_offs}[i+1]]. \tag{19}$$

This representation uses $O(m + \text{nnz})$ storage.

CSR output uses the same row structure, but adds a value array `vals` of type `[nnz]f64`. Here, `vals` has the same length as `row_idx`. For each row, the entries in `vals` are stored in the same order as the corresponding column indices in `row_idx`. As such, the slice for row $i$ in `vals` contains the values for the column indices in the same slice of `row_idx`.

For the running example, the sparsity pattern is given in Eq. (6). Its row-wise CSR pattern representation is

$$\texttt{row\_offs} = [0, 2, 4, 6, 8], \qquad \texttt{row\_idx} = [0, 2,\ 1, 3,\ 2, 3,\ 0, 1]. \tag{20}$$

For example, row 0 is given by $\texttt{row\_idx}[\texttt{row\_offs}[0] : \texttt{row\_offs}[1]] = \texttt{row\_idx}[0 : 2]$, so its nonzero column indices are $[0, 2]$. Similarly, row 1 is given by $\texttt{row\_idx}[2 : 4]$, giving the nonzero column indices $[1, 3]$.

If the corresponding nonzero Jacobian values are stored in CSR output form, then the value array is

$$\texttt{vals} = [\cos(x_0),\ 3x_2^2,\ x_3,\ x_1,\ 2x_2,\ 3,\ x_1,\ x_0]. \tag{21}$$

For instance, row 0 uses the slice $[0 : 2]$. The corresponding column indices are $[0, 2]$, and the corresponding values are $[\cos(x_0), 3x_2^2]$. These represent $J_f(x)_{0,0} = \cos(x_0)$ and $J_f(x)_{0,2} = 3x_2^2$.

The coloring algorithms also need the opposite direction: for each column, they need the rows where structurally nonzero entries may occur. The library therefore also stores a column-wise representation using two arrays, `col_offs` of type `[n+1]i64` and `col_idx` of type `[nnz]i64`, where `col_offs` stores where each column begins and `col_idx` stores the row indices of the structurally nonzero entries. For the same example,

$$\texttt{col\_offs} = [0, 2, 4, 6, 8], \qquad \texttt{col\_idx} = [0, 3,\ 1, 3,\ 0, 2,\ 1, 2]. \tag{22}$$

Using the same bipartite graph notation as in Figure 1, $r_i$ denotes row $i$ and $c_j$ denotes column $j$. The same graph is represented here using the row-wise and column-wise arrays used by the implementation. The two views are shown in Table 1.

| row vertex | neighboring columns |
|---|---|
| $r_0$ | $c_0,\ c_2$ |
| $r_1$ | $c_1,\ c_3$ |
| $r_2$ | $c_2,\ c_3$ |
| $r_3$ | $c_0,\ c_1$ |

| column vertex | neighboring rows |
|---|---|
| $c_0$ | $r_0,\ r_3$ |
| $c_1$ | $r_1,\ r_3$ |
| $c_2$ | $r_0,\ r_2$ |
| $c_3$ | $r_1,\ r_2$ |

Table 1: Row-wise (left) and column-wise (right) views of the running example.

As shown in Table 1, the row-wise representation to the left gives the neighboring columns of each row, while the column-wise representation to the right gives the neighboring rows of each column. Together, these two representations store the bipartite graph structure needed by the coloring algorithms. This is consistent with sparse derivative coloring software such as ColPack, which also stores graph and bipartite graph data structures in CSR format [3].

## 3.3 Overall Sparse Jacobian Pipeline

Given the CSR/bipartite sparsity pattern representation from the previous subsection, the library computes numerical Jacobian values through the following pipeline:

| | |
|---|---|
| CSR/bipartite sparsity pattern | (Section 3.2) |
| ↓ | |
| Preparation phase: color rows or columns | (Section 3.4) |
| ↓ | |
| Evaluation phase: compressed JVP or VJP at $x$ | (Section 3.5) |
| ↓ | |
| Return compressed, CSR, or dense output | (Sections 3.5 and 3.7) |

The preparation phase depends only on the sparsity pattern. In the JVP version, the columns of the Jacobian pattern are colored, while in the VJP version, the rows are colored. The purpose of coloring is to determine which columns or rows can be evaluated together without overlapping structurally nonzero entries. Since this phase only uses structural information, it does not depend on the particular input point $x$.

The evaluation phase uses the prepared representation together with $f$ and $x$. It performs one compressed JVP or VJP evaluation per color. The compressed result can either be returned directly, or reconstructed into CSR or dense output. CSR variants such as `eval_prepared_jvp_csr` and `eval_prepared_vjp_csr` reconstruct the final `vals` array, while dense variants expand the sparse values into a full matrix.

This separation makes it possible to reuse the same prepared representation for several input points, as long as the sparsity pattern stays fixed. The same structure is used in automatic mode. Functions such as `prepare_jac_auto` compute both the column and row colorings and compare their number of colors, while `eval_prepared_auto_csr` evaluates the selected mode and returns CSR output.

The following subsections explain the coloring step, the JVP and VJP evaluation methods, automatic mode selection, reusable preprocessing, and the output formats in more detail.

## 3.4 Coloring the Sparsity Pattern

Coloring is the pattern-dependent preprocessing step that groups structurally non-conflicting rows or columns. Since coloring depends only on the sparsity pattern, the resulting color arrays can be reused as long as the pattern is unchanged.

For JVP-based compression, the library colors the column vertices $C$. Columns with the same color form one group $C_k$, which is later used to construct a compressed JVP seed vector. For VJP-based compression, the same idea is applied to the row vertices $R$.

The conflict rule is based on distance-2 paths in the bipartite graph. For column coloring, two columns conflict if they share a row vertex, or equivalently if they are connected by a path of the form column–row–column. Using Table 1, column $c_0$ is adjacent to rows $r_0$ and $r_3$. These rows are adjacent to columns $c_0, c_2$ and $c_0, c_1$, respectively. Ignoring $c_0$ itself, the distance-2 neighbors of $c_0$ are therefore $c_1$ and $c_2$, so $c_0$ cannot receive the same color as either of them.

For row coloring, the same rule is applied with paths of the form row–column–row.

The library contains two coloring implementations. The first is a greedy partial distance-2 coloring implementation, referred to as `D2` in the evaluation. The second is an optimistic vertex-based bipartite-graph partial coloring implementation, referred to as `BGPC` in the evaluation. Both solve the same partial distance-2 coloring problem described above, but D2 uses a sequential greedy coloring loop, whereas BGPC uses an optimistic work-queue structure intended to be more parallel.

The D2 implementation follows Algorithm 3.2 in *What Color Is Your Jacobian?* [4]. Algorithm 1 shows the same algorithm, but uses zero-based colors to match the implementation. The algorithm is stated for a general bipartite graph

$$G_b = (V_1, V_2, E), \tag{23}$$

where $V_2$ is the side to be colored and $V_1$ is the opposite side used to detect conflicts.

**Algorithm 1** Greedy partial distance-2 coloring

**procedure** PARTIALD2COLORINGALG($G_b = (V_1, V_2, E)$)
Let $v_1, \ldots, v_{|V_2|}$ be a given ordering of $V_2$
Initialize `forbiddenColors` with some value $a \notin V_2$
**for** $i \leftarrow 1$ to $|V_2|$ **do**
  **for** each $w \in N_1(v_i)$ **do**
    **for** each colored vertex $x \in N_1(w)$ **do**
      `forbiddenColors`[`color`[$x$]] $\leftarrow v_i$
    **end for**
  **end for**
  `color`[$v_i$] $\leftarrow \min\{c \geq 0 \mid$ `forbiddenColors`[$c$] $\neq v_i\}$
**end for**
**end procedure**

For column coloring, the D2 algorithm is instantiated with $V_1 = R$ and $V_2 = C$. For row coloring, the roles are swapped, so $V_1 = C$ and $V_2 = R$. The same algorithm can therefore be used for both sparse JVP and sparse VJP computation. In the code, these two cases are implemented by functions such as `partial_d2_color_cols` and `partial_d2_color_rows`. The `forbiddenColors` array is timestamped by the current iteration index to avoid clearing it for every vertex.

The BGPC implementation is provided by functions such as `vv_color_cols` and `vv_color_rows`. It was added because the greedy D2 implementation contains a sequential coloring loop and is therefore not well suited for a full GPU pipeline. The implementation is based on the optimistic vertex-based bipartite-graph partial coloring structure of Taş, Kaya, and Saule [10]. The outer work-queue loop follows their Algorithm 1, while the coloring and conflict-removal phases correspond to their Algorithms 4 and 5. The BGPC implementation is adapted to Futhark and is included as an alternative coloring method for comparison in the evaluation, while D2 is used as the default in the sparse Jacobian library because it was the most robust coloring method in the benchmarks.

## 3.5 Sparse Jacobian via JVP and VJP

After coloring, sparse Jacobian evaluation is point-dependent. The direct output of the AD evaluation is compressed: one derivative result is computed for each color. The sparsity pattern and color arrays determine the seed vectors and how the compressed results can be unpacked, while the point $x$ determines the derivative values. In this subsection, color numbers and array indices follow Futhark's zero-based indexing.

### 3.5.1 Compressed JVP and VJP Evaluation

For the JVP pipeline, the array `colors` of type `[n]i64` maps each Jacobian column to a zero-based color. For each column color $k$, where $0 \leq k <$ `num_col_colors`, the implementation constructs a seed vector $s_k \in \{0,1\}^n$, with $s_k[j] = 1$ exactly when `colors[j]` $= k$, and $s_k[j] = 0$ otherwise. It then computes one compressed JVP result for that color:

$$z_k = \texttt{jvp}\ f\ x\ s_k = J_f(x)s_k. \tag{24}$$

Each $z_k$ is a vector of length $m$. Its entry $z_k[i]$ contains the contribution to row $i$ from the columns with color $k$. The collection of all $z_k$ vectors is the compressed JVP output.

The VJP pipeline uses the same idea on rows instead of columns. The array `row_colors` of type `[m]i64` maps each Jacobian row to a zero-based color. For each row color $k$, where $0 \leq k <$ `num_row_colors`, the implementation constructs a row seed vector $\bar{s}_k \in \{0,1\}^m$, with $\bar{s}_k[i] = 1$ exactly when `row_colors[i]` $= k$, and $\bar{s}_k[i] = 0$ otherwise. It then computes one compressed VJP result for that row color:

$$\bar{z}_k = \texttt{vjp}\ f\ x\ \bar{s}_k = \bar{s}_k^T J_f(x). \tag{25}$$

Each $\bar{z}_k$ is a vector of length $n$. Its entry $\bar{z}_k[j]$ contains the contribution to column $j$ from the rows with color $k$. The collection of all $\bar{z}_k$ vectors is the compressed VJP output.

### 3.5.2 CSR Reconstruction

The compressed output is not yet in CSR format. CSR output is obtained by filling a value array `vals` for the known sparsity pattern. The arrays `row_offs` and `row_idx` are already fixed by the pattern, so reconstruction only has to compute the values.

The reconstruction is done row by row. For a row $i$, the structurally nonzero entries are stored at CSR positions

$$\texttt{row\_offs[i]} \leq t < \texttt{row\_offs[i+1]}. \tag{26}$$

For each such position $t$, the corresponding column is

$$j = \texttt{row\_idx[t]}. \tag{27}$$

As such, `vals[t]` must contain the Jacobian entry $J_f(x)_{ij}$.

For JVP reconstruction, the column $j$ determines which color to read from:

$$k = \texttt{colors[j]}. \tag{28}$$

As shown in Eq. (17), the compressed JVP output can be viewed as a matrix with one column per color. In the implementation, it is stored color-first. Writing this output as `ys_jvp`, the CSR value is

$$\texttt{vals[t]} = \texttt{ys\_jvp[k][i]}. \tag{29}$$

This gives $J_f(x)_{ij}$, because the column coloring ensures that no other column with color $k$ contributes to row $i$.

For VJP reconstruction, the same rule is used with rows and columns swapped. Writing the compressed VJP output as `ys_vjp`, the row color $k =$ `row_colors[i]` selects the compressed VJP result, and the CSR value is

$$\texttt{vals[t]} = \texttt{ys\_vjp[k][j]}. \tag{30}$$

This gives the same entry $J_f(x)_{ij}$, because the row coloring ensures that no other row with color $k$ contributes to column $j$.

Both pipelines can therefore return CSR output in the same format,

$$(\texttt{row\_offs}, \texttt{row\_idx}, \texttt{vals}). \tag{31}$$

### 3.6 Automatic Mode Selection and Reuse

The implementation supports both a JVP-based and a VJP-based sparse Jacobian pipeline. The automatic pipeline chooses between these two modes and returns the result in CSR format. In this subsection, the preparation phase refers to the pattern-dependent work performed after a CSR/bipartite sparsity pattern is available.

For a given CSR pattern, the automatic preparation computes both a column coloring and a row coloring. The column colors are used by the JVP pipeline, and the row colors are used by the VJP pipeline. The implementation then compares the two color counts:

$$\texttt{use\_jvp} = \texttt{num\_col\_colors} \leq \texttt{num\_row\_colors}. \tag{32}$$

If `use_jvp` is true, evaluation uses the JVP pipeline. Otherwise, it uses the VJP pipeline. The tie case is handled deterministically by choosing JVP.

This is a simple heuristic. Fewer column colors means fewer JVP calls, while fewer row colors means fewer VJP calls. The automatic pipeline therefore chooses the mode with the fewest colors. This does not guarantee the fastest runtime in all cases, since the relative cost of a JVP and a VJP may depend on the function and the backend. However, the choice is simple, pattern-dependent, and can be made during preparation.

The automatic pipeline uses CSR as its common sparse output format. It does not return the compressed output directly, because the compressed JVP and VJP outputs have different shapes and indexing conventions. The JVP output contains one vector per column color, while the VJP

output contains one vector per row color. Both compressed forms are therefore reconstructed into the same CSR format:

$$(\texttt{row\_offs},\ \texttt{row\_idx},\ \texttt{vals}). \tag{33}$$

The prepared representation stores the CSR structure, the column and row color arrays, the two color counts, and `use_jvp`. These values depend only on the sparsity pattern, not on the evaluation point $x$, and can therefore be reused across several Jacobian evaluations with the same pattern. During evaluation, the prepared representation is combined with $f$ and $x$, and only the derivative values are recomputed for the current point. This is useful in settings such as Newton's method applied to a problem where the point changes between iterations, but the sparsity pattern stays the same.

### 3.7 Dense Baselines and Convenience Outputs

The implementation distinguishes between the direct compressed output of the sparse pipelines and reconstructed output formats. The compressed output is closest to the actual AD work where one derivative result is computed per color. CSR output and dense output are then obtained by placing these values back into the sparsity pattern.

Figure 2 shows the three output formats for the running column-coloring example. The compressed output has one column per color. CSR output stores the structurally nonzero values using the row-wise CSR structure from the sparsity pattern. Dense output stores the same Jacobian as a full matrix, with zeros inserted at structurally zero positions.

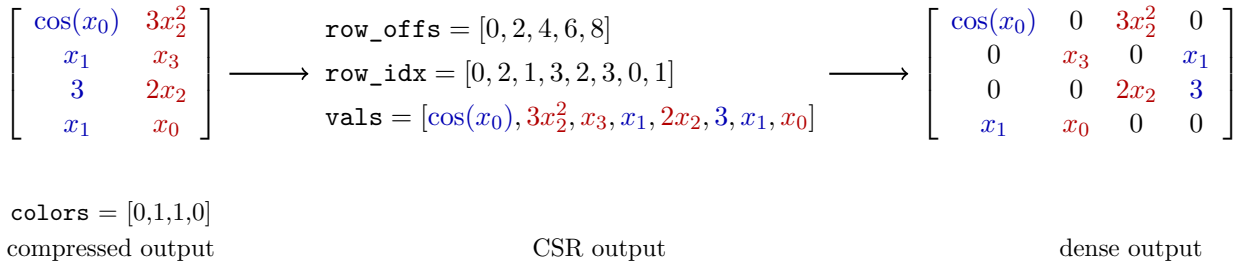



Figure 2: Compressed, CSR, and dense output formats for the running example.

The compressed output is the direct result of the sparse AD evaluation. It is the main output of the individual JVP and VJP sparse pipelines, but its shape depends on the coloring and on which pipeline is used.

CSR output is a reconstructed sparse format. It is especially useful for the automatic pipeline, because the JVP and VJP compressed outputs have different shapes, while both can be reconstructed into the same CSR structure.

Dense output is only a convenience format. It expands the sparse values to a full $[m][n]$ matrix by placing computed derivative values at structurally nonzero positions and inserting zeros elsewhere. This is useful for small examples and visual inspection, but it is not intended as the main sparse output. It should also not be confused with the dense baseline in the benchmarks, which computes the full Jacobian directly.

### 3.8 Work, Span, and Space Complexity

This subsection summarizes the work, span, and space complexity of the sparse Jacobian pipeline. Work is the total amount of computation, while span is the longest chain of sequential dependencies in the usual Futhark sense [6]. Table 2 lists the main operations. Coloring is separated from compressed AD evaluation because the coloring algorithm can be exchanged.

Here, $m$ and $n$ are the number of output and input components, and nnz is the number of structurally nonzero Jacobian entries. The quantities $d_{\text{row}}$ and $d_{\text{col}}$ are the maximum number of nonzeros in any row and column, respectively. The number of column colors is $p$, and the number of row colors is $q$. For BGPC, $Q$ is the current work queue, and $b$ is the size of the local forbidden-color array used per active vertex.

We write $W(f)$ and $S(f)$ for the work and span of evaluating $f$ once. A JVP or VJP computes one Jacobian-vector or vector-Jacobian product by executing $f$ with AD [8]. We therefore count one such pass as $O(W(f))$ work and $O(S(f))$ span, ignoring constant-factor AD overheads. Since the seed directions are independent, more colors multiply the work but not the span.

| Pipeline operation | Work | Span | Space |
|---|---|---|---|
| **CSR from dense pattern**<br>`csr_rows_from_pattern`<br>`csr_cols_from_pattern` | $O(mn)$ | $O(\log(mn))$ | $O(mn)$ |
| **D2 column**<br>`partial_d2_color_cols` | $O(n + \text{nnz} \cdot d_{\text{row}})$ | $O(n + \text{nnz} \cdot d_{\text{row}})$ | $O(n)$ |
| **D2 row**<br>`partial_d2_color_rows` | $O(m + \text{nnz} \cdot d_{\text{col}})$ | $O(m + \text{nnz} \cdot d_{\text{col}})$ | $O(m)$ |
| **BGPC column, per iteration**<br>`vv_color_cols` | $O(\text{nnz} \cdot d_{\text{row}} + \lvert Q\rvert b)$ | $O(d_{\text{col}} d_{\text{row}} + b + \log n)$ | $O(n + \lvert Q\rvert b)$ |
| **BGPC row, per iteration**<br>`vv_color_rows` | $O(\text{nnz} \cdot d_{\text{col}} + \lvert Q\rvert b)$ | $O(d_{\text{row}} d_{\text{col}} + b + \log m)$ | $O(m + \lvert Q\rvert b)$ |
| **Compressed JVP**<br>`compressed_ys_jvp` | $O(pn + p \cdot W(f))$ | $O(\log n + S(f))$ | $O(p(m+n))$ |
| **Compressed VJP**<br>`compressed_ys_vjp` | $O(qm + q \cdot W(f))$ | $O(\log m + S(f))$ | $O(q(m+n))$ |
| **CSR value reconstruction**<br>`compressed_to_csr_vals` | $O(m + \text{nnz})$ | $O(m)$ | $O(\text{nnz})$ |

Table 2: Work, span, and space complexity of the main sparse Jacobian pipeline operations. CSR construction and CSR value reconstruction are optional operations around the main compressed-output pipeline.

The $O(pn)$ and $O(qm)$ terms come from constructing full-length seed vectors. Compressed JVP constructs $p$ seeds of length $n$, giving $O(pn)$, while compressed VJP constructs $q$ seeds of length $m$, giving $O(qm)$. Dense JVP and dense VJP can be seen as the special cases $p = n$ and $q = m$. Therefore, coloring reduces the number of AD passes, but each seed is still represented as a full-length array.

For the main compressed-output pipelines, CSR construction from a dense pattern and CSR

value reconstruction are not included. These are optional operations around the sparse pipeline. We use $W(\text{color})$ and $S(\text{color})$ for the work and span of the coloring step used by the chosen pipeline. For JVP this is column coloring, and for VJP this is row coloring. The sparse JVP pipeline has

$$\textbf{Work: } O(W(\text{color}) + pn + p \cdot W(f)) \quad \textbf{Span: } O(S(\text{color}) + \log n + S(f)). \tag{34}$$

The sparse VJP pipeline has

$$\textbf{Work: } O(W(\text{color}) + qm + q \cdot W(f)) \quad \textbf{Span: } O(S(\text{color}) + \log m + S(f)). \tag{35}$$

These formulas show what changes when the coloring algorithm is replaced. The seed-construction and AD-evaluation terms stay the same. Only $W(\text{color})$ and $S(\text{color})$ change. With the D2 implementation, these coloring terms are $O(n + \text{nnz} \cdot d_{\text{row}})$ for JVP and $O(m + \text{nnz} \cdot d_{\text{col}})$ for VJP. The $\text{nnz} \cdot d_{\text{row}}$ and $\text{nnz} \cdot d_{\text{col}}$ terms follow from the greedy partial distance-2 coloring bound $O(|E|\Delta(V_1))$ given by Gebremedhin et al. [4]. The extra $n$ and $m$ terms come from initializing arrays and from the sequential outer loop over the colored side, including vertices with no nonzeros. With BGPC, the table reports per-iteration upper bounds based on the bounds given by Taş et al. [10]. The total coloring cost is the sum of these bounds over all iterations until the work queue is empty.

For CSR construction from a dense pattern, the table reports $O(mn)$ space because the implementation creates dense temporary arrays of size $mn$. The final row-wise CSR pattern itself only needs $O(m + \text{nnz})$ space. CSR value reconstruction loops over all rows and, within each row, over the nonzeros in that row. This gives $O(m)$ span and $O(m + \text{nnz})$ work.

In automatic mode, preparation computes both a column coloring and a row coloring. The preparation cost is therefore the sum of the two coloring costs for the chosen coloring algorithm. Evaluation then chooses between compressed JVP and compressed VJP according to the number of colors. Since automatic mode returns CSR, CSR value reconstruction adds the cost shown in Table 2. The sparse pipeline is most useful when $p$ is much smaller than $n$ or $q$ is much smaller than $m$, and the saved AD work outweighs the coloring overhead.

# 4 Evaluation

## 4.1 Evaluation Goals and Setup

The evaluation has two main goals. First, it validates that the sparse Jacobian implementations compute the same structurally nonzero values as the dense baselines. Second, it measures whether exploiting a known sparsity pattern reduces runtime compared to dense Jacobian computation.

For correctness, the sparse outputs are compared with the corresponding structurally nonzero entries of the dense Jacobian. This checks that the sparse pipelines recover the same derivative values for the entries marked by the sparsity pattern.

The evaluation uses four benchmark problems. `Banded5` and `Stencil` are synthetic benchmarks with controlled sparsity patterns, included to test predictable sparse structures and scaling behavior. Bundle Adjustment (`BA`) and Hand Tracking (`HT`) are based on GradBench workloads, originally from ADBench, and are used to evaluate the sparse pipelines on more realistic AD programs [9, 5]. The problem-specific `calculate_jacobian` implementations used for comparison are also taken from these benchmark sources.

The benchmark results are reported as speedups relative to the dense CPU baseline:

$$\text{speedup} = \frac{T_{\text{Dense CPU}}}{T_{\text{method}}}. \tag{36}$$

Larger values are better, and a value above $1\times$ means that the method is faster than dense Jacobian computation on the CPU. The full runtimes in microseconds are reported in Appendix 6.

The dense baselines return full dense Jacobians. The sparse benchmark pipelines use the compressed sparse values produced directly by the compressed AD calls. CSR reconstruction is included only where the benchmarked method produces CSR output, such as the `BA calculate_jacobian` benchmark.

All benchmarks were run using `futhark bench --runs=10` with either the multicore or CUDA backend. In the result tables, CPU refers to the Futhark multicore backend, while GPU refers to the Futhark CUDA backend. The benchmarks were run through Slurm on the Hendrix cluster. The final benchmark jobs requested 64 CPU cores using `--cpus-per-task=64`. GPU benchmarks were run on one NVIDIA A100 80GB PCIe GPU.

Before running the tests or benchmarks, package dependencies should be installed with `futhark pkg sync` in the repository root, `benchmark/ba`, and `benchmark/ht`. The tests can be run with `make test`, assuming that Futhark and CUDA are installed or loaded and an NVIDIA GPU is available. The individual test files are listed in Table 3. The reported benchmark groups can be run with `make bench` under the same assumption.

## 4.2 Correctness Testing

The correctness tests are organized by implementation layer, following the main parts of the sparse Jacobian pipeline. This makes the tests easier to interpret, since errors can occur in several different places: sparsity representation, CSR construction, coloring, compressed AD evaluation, and sparse output reconstruction.

For the sparse Jacobian tests, the reference is the dense Jacobian restricted to the known sparsity pattern. Sparse outputs are expanded to dense matrices with zeros outside the sparsity pattern. For every structurally nonzero position $(i, j)$, the floating-point tests check that

$$|J_{\text{sparse}}(i, j) - J_{\text{dense}}(i, j)| \leq 10^{-9}. \tag{37}$$

Entries outside the sparsity pattern are set to zero in both matrices before comparison, so the tests only check derivative values at structurally nonzero positions. CSR construction and coloring tests use exact checks, since they work with boolean patterns and integer color assignments rather than floating-point derivative values.

| Layer and test files | Tests and edge cases |
|---|---|
| **CSR representation**<br>`test_pattern_csr` | normal, empty, dense, diagonal, empty-row, and bipartite patterns |
| **Coloring**<br>`test_partial_d2_coloring`<br>`test_bgpc_vv_coloring` | mixed, empty, diagonal, dense, and star patterns, with row and column validity |
| **Dense baselines**<br>`test_dense_jacobian` | wide, square, tall, and constant functions, with analytic Jacobians |
| **Sparse JVP/VJP pipelines**<br>`test_sparse_jacobian_jvp`<br>`test_sparse_jacobian_vjp` | zero pattern, nonlinear mixed pattern, empty row, explicit colors, compressed output, CSR output, prepared reuse, and CSR input |
| **Automatic pipeline**<br>`test_sparse_jacobian_auto` | JVP choice, VJP choice, tie case, zero pattern, and CSR input |
| **Benchmark pipelines**<br>`test_jvp_ba_correctness`<br>`test_jvp_ht_correctness` | small BA and HT instances with D2 and BGPC coloring |

Table 3: Correctness test coverage by implementation layer.

Table 3 summarizes the test coverage. The CSR tests check that boolean sparsity patterns are converted correctly to row-wise and column-wise CSR structures. The coloring tests check the validity condition required for compression: columns with the same color must not occur in the same Jacobian row. The row-coloring tests check the corresponding condition for rows.

The dense baseline tests compare dense JVP and dense VJP reconstruction against analytic Jacobians. The sparse JVP and VJP tests then compare compressed and CSR outputs against the dense Jacobian restricted to the sparsity pattern. These tests also cover prepared reuse, explicit color assignments, and the CSR input.

The automatic pipeline tests check that the implementation selects the expected mode from the number of row and column colors. They also cover the tie case, where the implementation chooses JVP deterministically. Finally, the BA and HT correctness tests validate the benchmark wrappers on small problem instances with both D2 and BGPC coloring.

These tests do not prove correctness for every possible input program. However, they test each implementation layer independently and also validate the full sparse pipelines used in the benchmark evaluation. All tests listed in Table 3 pass.

## 4.3 Benchmark Problems

### 4.3.1 Banded5

The `Banded5` benchmark has exactly five structurally nonzero entries per row. Each output element depends on a center input element and its two nearest neighbors on each side. The indices wrap around at the boundary. Since the input vector is wider than the output vector in the tested cases, the benchmark uses a stride $s = n/m$, so row $i$ is centered at input index $i \cdot s$.

For example, a small $5 \times 10$ sparsity pattern with stride $s = 2$ is:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}. \tag{38}$$

The actual benchmark matrices are much wider, so the Jacobian is highly sparse. The five nonzeros per row also make this a low-color benchmark: both D2 and BGPC use five column colors for all tested sizes.

### 4.3.2 Stencil

`Stencil` is a synthetic two-dimensional benchmark based on a five-point stencil with wrap-around boundaries. The input vector is interpreted as an $h \times w$ grid, and the function returns one output value for each grid point. Each output value depends on the input value at the same grid point and the input values at its north, south, east, and west neighbors. At the boundary, the neighbor indices wrap around to the other side of the grid.

For one output grid point, the dependency pattern in the surrounding grid is:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}. \tag{39}$$

This pattern is used for every grid point, so each row of the Jacobian has five structurally nonzero entries. Unlike `Banded5`, the nonzeros are arranged according to a two-dimensional grid rather than a one-dimensional band. This makes `Stencil` a structured sparse benchmark with a different geometry than the one-dimensional `Banded5` case.

### 4.3.3 Bundle Adjustment

`BA` is the bundle adjustment benchmark from GradBench, originally based on ADBench [9]. The benchmark models a bundle adjustment problem where camera parameters and 3D point positions are optimized from image observations. Each observation relates one camera to one 3D point and contributes two image residuals and one weight residual.

We let $C$ be the number of cameras, $M$ the number of 3D points, and $N$ the number of observations. Each camera has 11 parameters, each point has 3 coordinates, and each observation has one scalar weight. The Jacobian therefore has

$$3N \times (11C + 3M + N) \tag{40}$$

entries in dense form. However, each observation only depends on the camera, point, and weight used by that observation. The two image residuals therefore have $11 + 3 + 1 = 15$ structurally nonzero entries each, while the weight residual only depends on its own weight.

Figure 3 shows a representative example of the resulting block sparsity pattern. The exact pattern depends on the observation list, since different observations may use different cameras and points.

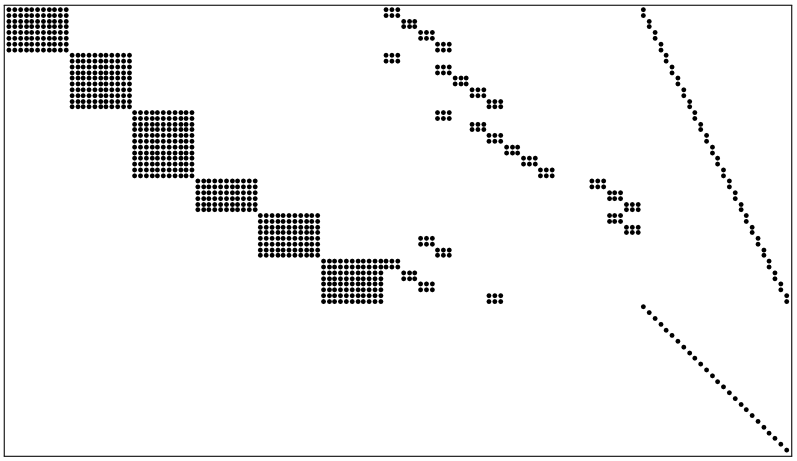


Figure 3: Representative block sparsity pattern of the `BA` Jacobian, adapted from the ADBench benchmark paper [9]. Black entries are structurally nonzero, while white entries are zero.

Column coloring is the better choice for the tested `BA` instances. Each observation touches only a small number of input variables, and both D2 and BGPC use 15 column colors. Row coloring is less effective because output rows can share input columns: rows from the same observation share the observation weight, and rows from different observations may share camera or point parameters. Therefore, `BA` is evaluated with compressed JVPs.

### 4.3.4 Hand Tracking

`HT` is the hand tracking benchmark from GradBench, originally based on ADBench [9]. The benchmark uses a fixed 3D hand model and a set of measured 3D points. Given motion parameters, the hand model is transformed into a posed model, and the objective compares the resulting model points with the measured points.

The differentiable inputs are the motion parameters $p \in \mathbb{R}^{26}$ and the variables $U \in \mathbb{R}^{2\times N}$. In the GradBench implementation, the motion parameters are called `theta`. The $U$ variables contain two variables for each measured point. The objective returns one three-dimensional residual for each measured point, so the Jacobian has

$$3N \times (26 + 2N) \tag{41}$$

entries in dense form.

Figure 4 shows a representative example of the resulting sparsity pattern. The first 26 columns correspond to the motion parameters. In our benchmark pattern, these columns are treated as structurally nonzero for every residual row. The remaining columns correspond to

the local $U$ variables. Each residual $q$ depends only on its own two variables $u_{q,1}$ and $u_{q,2}$, and not on the $U$ variables of other measured points. Thus, each residual row has $26 + 2 = 28$ structurally nonzero entries.

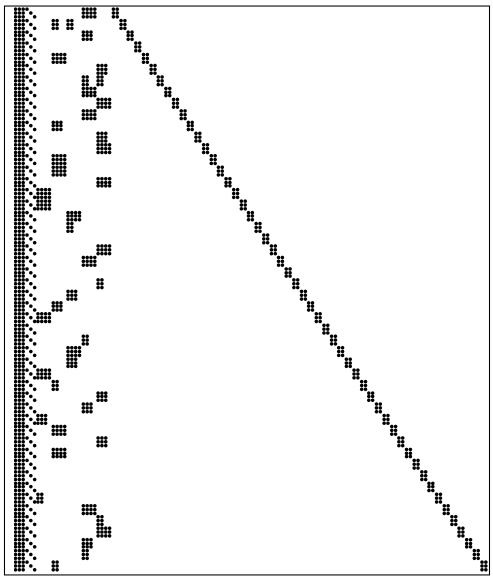


Figure 4: Representative sparsity pattern of the `HT` Jacobian, adapted from the ADBench benchmark paper [9]. Black entries are structurally nonzero, while white entries are zero.

In the sparsity pattern used for `HT`, every residual row marks the 26 motion-parameter columns as structurally nonzero. Therefore, any two rows share structurally nonzero columns, so row coloring would require one color per residual row. Column coloring is therefore the better choice. In the tested `HT` instances, both D2 and BGPC use 28 column colors, so `HT` is evaluated with compressed JVPs.

## 4.4 End-to-End Performance

This section reports end-to-end performance for the dense baselines and the full sparse pipelines. The Dense CPU column gives the baseline runtime in microseconds, while all other columns report speedups relative to Dense CPU. The sparse timings include coloring and compressed AD evaluation and use compressed output rather than CSR reconstruction. The full runtimes are reported in Appendix 6.

Table 4 shows the clearest benefit from sparsity. For `Banded5`, D2 CPU is between 30.6× and 217.9× faster than Dense CPU, while BGPC GPU also gives large speedups and is fastest on the largest instance, reaching 438.4×. For `Stencil`, D2 CPU gives the largest speedups, reaching 208.5× on the largest instance. In contrast, BGPC CPU performs poorly on `Stencil`, while BGPC GPU is faster than Dense CPU only on the largest `Stencil` case and remains much slower than D2 CPU. As discussed in the next subsection, D2 and BGPC use the same or nearly the same number of colors for these cases, so the runtime difference is mainly caused by coloring cost and backend behavior rather than by the number of compressed AD calls.

| Case | Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|---|
| Banded5 | $512 \times 16384$ | 15651 | 4.5× | 30.6× | 4.4× | 25.1× |
| Banded5 | $1024 \times 32768$ | 74388 | 5.1× | 120.0× | 16.3× | 114.3× |
| Banded5 | $2048 \times 65536$ | 295952 | 5.2× | 217.9× | 47.9× | 438.4× |
| Stencil | $4096 \times 4096$ | 23377 | 17.2× | 33.3× | 0.7× | 0.9× |
| Stencil | $9216 \times 9216$ | 86072 | 12.7× | 102.1× | 0.3× | 0.9× |
| Stencil | $16384 \times 16384$ | 260597 | 11.1× | 208.5× | 0.4× | 1.4× |

Table 4: End-to-end JVP results on `Banded5` and `Stencil`. Dense CPU is reported in microseconds. All other columns are speedups relative to Dense CPU, so values below 1× mean that the method is slower than Dense CPU.

The VJP results in Table 5 test the opposite AD direction on the structured benchmarks. The VJP benchmarks use different `Banded5` sizes than the JVP benchmarks, so the two tables should not be compared size-for-size. D2 CPU gives clear speedups on both structured problems and is faster than Dense GPU for most tested cases, while BGPC performs poorly overall. The VJP row-color counts are the same for D2 and BGPC on `Banded5`. On `Stencil`, D2 uses two fewer colors on one case, which is unlikely to explain the much larger gap in runtime.

| Case | Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|---|
| Banded5 | $2048 \times 4096$ | 1127 | 2.7× | 2.5× | 0.04× | 0.02× |
| Banded5 | $4096 \times 8192$ | 3891 | 2.5× | 7.0× | 0.03× | 0.03× |
| Banded5 | $8192 \times 16384$ | 18022 | 2.7× | 26.3× | 0.03× | 0.06× |
| Stencil | $4096 \times 4096$ | 3721 | 1.8× | 1.9× | 0.09× | 0.15× |
| Stencil | $9216 \times 9216$ | 18079 | 1.6× | 5.6× | 0.07× | 0.18× |
| Stencil | $16384 \times 16384$ | 57130 | 1.4× | 45.5× | 0.09× | 0.3× |

Table 5: End-to-end VJP results on `Banded5` and `Stencil`. Dense CPU is reported in microseconds. All other columns are speedups relative to Dense CPU.

The `BA` and `HT` results in Table 6 are more mixed, but still show substantial sparse speedups. On `BA`, D2 CPU is faster than Dense CPU for all tested sizes and faster than Dense GPU on the three largest sizes. Dense GPU is fastest on the smallest `BA` case. BGPC GPU is slower than both Dense GPU and D2 CPU on these runs. On `HT`, D2 CPU is faster than Dense CPU and Dense GPU for all tested sizes.

| Case | Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|---|
| BA | $6144 \times 2784$ | 20566 | 22.9× | 15.2× | 2.5× | 1.4× |
| BA | $12288 \times 5200$ | 54673 | 17.6× | 22.3× | 4.6× | 2.4× |
| BA | $24576 \times 9664$ | 118479 | 10.9× | 21.8× | 5.4× | 3.2× |
| BA | $36864 \times 14128$ | 210175 | 8.4× | 29.4× | 7.5× | 4.1× |
| HT | $1536 \times 1050$ | 17756 | 0.7× | 1.4× | 0.7× | 0.04× |
| HT | $3072 \times 2074$ | 59801 | 0.7× | 3.0× | 1.2× | 0.07× |
| HT | $6144 \times 4122$ | 242386 | 0.7× | 6.8× | 2.5× | 0.14× |
| HT | $12288 \times 8218$ | 903892 | 0.6× | 15.2× | 4.7× | 0.3× |

Table 6: End-to-end JVP results on `BA` and `HT`. Dense CPU is reported in microseconds. All other columns are speedups relative to Dense CPU.

Across all benchmarks, D2 CPU is the strongest CPU-based sparse variant and the most robust sparse pipeline overall. BGPC CPU is faster than Dense CPU on some benchmarks, but it is generally slower than D2 CPU. BGPC GPU is more benchmark-dependent. It performs very well on `Banded5` JVP, especially on the largest instance, but it is not competitive with D2 CPU on `BA`, `HT`, `Stencil`, or the VJP benchmarks. The next subsection separates coloring from precolored evaluation to explain where these differences come from on CPU.

### 4.5 Pipeline Breakdown

The end-to-end timings show the total runtime of the sparse pipeline, but not how that time is distributed across the pipeline steps. To estimate this, we benchmark coloring separately and also run a precolored D2 variant where the color assignment is supplied as input. The precolored variant skips coloring, but still constructs the compressed seeds, performs the compressed AD calls, and returns the compressed sparse output. These are separate benchmarks, so the numbers should be read as approximate indicators rather than as parts that add up exactly. The breakdown is reported for the CPU backend and for the JVP pipeline only.

Here, the number of colors is the number of compressed JVP seed vectors used to compute the compressed Jacobian. For `Banded5`, both D2 and BGPC use five colors for all tested sizes. For `Stencil`, D2 uses $9, 7, 9$ colors on the three tested sizes in increasing order, while BGPC uses nine colors for all three sizes. For `BA`, both methods use 15 colors for all tested sizes, and for `HT`, both methods use 28 colors for all tested sizes.

For `Banded5` and `Stencil` in Table 7, the D2 precolored timings stay below half a millisecond for all tested sizes, and D2 coloring is also small. BGPC coloring is much more expensive, especially for `Stencil`, where the coloring time alone is much larger than the full D2 pipeline even though BGPC produces almost the same number of colors as D2.

| Case | Size | D2 full | D2 color | D2 precolored | BGPC color |
|---|---|---|---|---|---|
| Banded5 | $512 \times 16384$ | 603 | 40 | 475 | 706 |
| Banded5 | $1024 \times 32768$ | 731 | 185 | 453 | 3091 |
| Banded5 | $2048 \times 65536$ | 1085 | 405 | 467 | 4301 |
| Stencil | $4096 \times 4096$ | 553 | 128 | 416 | 29133 |
| Stencil | $9216 \times 9216$ | 746 | 278 | 422 | 220449 |
| Stencil | $16384 \times 16384$ | 972 | 505 | 444 | 654910 |

Table 7: Pipeline breakdown for `Banded5` and `Stencil` on the CPU backend. Times are reported in microseconds. The precolored D2 pipeline receives the color assignment as input.

Table 8 has the pipeline breakdown for `BA` and `HT`. For `BA`, D2 coloring is a significant part of the full D2 CPU runtime across all tested sizes, while the precolored D2 timings are much lower. For `HT`, the precolored D2 timings account for most of the full D2 runtime, which indicates that the post-coloring part of the sparse pipeline dominates more than coloring. In both benchmarks, BGPC coloring is substantially more expensive than D2 coloring.

| Case | Size | D2 full | D2 color | D2 precolored | BGPC color |
|---|---|---|---|---|---|
| BA | $6144 \times 2784$ | 1456 | 1047 | 429 | 7297 |
| BA | $12288 \times 5200$ | 2919 | 2105 | 432 | 10621 |
| BA | $24576 \times 9664$ | 5011 | 4285 | 1105 | 17925 |
| BA | $36864 \times 14128$ | 6831 | 6261 | 1562 | 25563 |
| HT | $1536 \times 1050$ | 12983 | 1062 | 8620 | 16063 |
| HT | $3072 \times 2074$ | 21587 | 2138 | 12249 | 34547 |
| HT | $6144 \times 4122$ | 30408 | 4277 | 23719 | 70228 |
| HT | $12288 \times 8218$ | 53996 | 8525 | 40801 | 135152 |

Table 8: Pipeline breakdown for `BA` and `HT` on the CPU backend. Times are reported in microseconds. The precolored D2 pipeline receives the color assignment as input.

Overall, the breakdown shows that D2 CPU is not faster because it finds fewer colors than BGPC. The color counts are usually the same or very close. The main difference on the CPU backend is that D2 computes these colorings much more quickly. This explains why D2 CPU is the strongest CPU-based sparse variant in Tables 4 and 6. The GPU end-to-end results are more mixed, since BGPC GPU is strong on `Banded5` JVP but much weaker on `BA`, `HT`, `Stencil`, and the VJP benchmarks. This suggests that backend behavior and the structure of the generated parallel work also matter, especially for BGPC.

### 4.6 Comparison with ADBench `calculate_jacobian`

As an additional reference point, we compare against the problem-specific `calculate_jacobian` implementations from the ADBench/GradBench benchmark sources. These implementations are reported separately because they are specialized to the individual benchmark problems, while our implementation uses the sparse Jacobian pipeline developed in this project. For `BA`, the comparison uses the D2 CSR pipeline, since the ADBench implementation also returns a CSR-like sparse output. For `HT`, the comparison uses the D2 compressed pipeline, which is closest to the output format used by the benchmark. As in the end-to-end tables, Dense CPU is reported in microseconds, while the remaining columns are speedups relative to Dense CPU.

For `BA` in Table 9, the problem-specific implementation is substantially faster than our D2 CSR pipeline in these measurements. This is especially clear on GPU, where the ADBench implementation reaches very large speedups relative to Dense CPU. This indicates that the benchmark-specific implementation is especially effective on the GPU backend.

| Jacobian size | Dense CPU | D2 CSR CPU | ADBench CPU | ADBench GPU |
|---|---|---|---|---|
| $6144 \times 2784$ | 20566 | 8.6× | 54.8× | 216.5× |
| $12288 \times 5200$ | 54673 | 13.8× | 57.9× | 563.6× |
| $24576 \times 9664$ | 118479 | 16.4× | 36.6× | 1184.8× |
| $36864 \times 14128$ | 210175 | 19.1× | 49.0× | 2040.5× |

Table 9: `BA` comparison with ADBench `calculate_jacobian`. Dense CPU is reported in microseconds. All other columns are speedups relative to Dense CPU.

For `HT` in Table 10, our D2 compressed pipeline on CPU is close to the problem-specific ADBench CPU implementation across the tested sizes. On GPU, however, the specialized

ADBench implementation is much faster. This shows that problem-specific structure can still matter a lot.

| Jacobian size | Dense CPU | D2 comp. CPU | ADBench CPU | ADBench GPU |
|---|---|---|---|---|
| $1536 \times 1050$ | 17756 | 1.3× | 1.0× | 20.1× |
| $3072 \times 2074$ | 59801 | 3.5× | 3.4× | 42.0× |
| $6144 \times 4122$ | 242386 | 8.2× | 8.8× | 96.1× |
| $12288 \times 8218$ | 903892 | 16.7× | 18.7× | 177.1× |

Table 10: `HT` comparison with ADBench `calculate_jacobian`. Dense CPU is reported in microseconds. D2 comp. CPU uses the compressed-output D2 pipeline. All other columns are speedups relative to Dense CPU.

## 4.7 Discussion

The results show that exploiting sparsity can substantially reduce Jacobian computation time in Futhark when the sparsity pattern is known. Across the benchmarks, D2 CPU is the strongest CPU-based sparse variant and the most robust sparse pipeline overall. It is faster than Dense CPU in all tested cases and faster than Dense GPU in most cases. This suggests that the sparse Jacobian pipeline can be useful even without writing benchmark-specific Jacobian code.

The main reason is that coloring reduces the number of AD calls. Dense JVP computes one direction per input component, while compressed JVP only needs one direction per color. This is effective for the tested benchmarks because the number of colors is small compared with the width of the Jacobian. For example, `BA` uses 15 colors and `HT` uses 28 colors for all tested sizes. As the dense Jacobian grows, this reduction becomes increasingly important, which is visible in the structured benchmarks and in the larger `BA` and `HT` cases.

The comparison between D2 and BGPC shows that the number of colors is not the only important factor. On CPU, D2 and BGPC usually find the same or nearly the same number of colors, and for several benchmarks these counts are already optimal, but D2 is much faster end-to-end. The pipeline breakdown shows that this mainly comes from coloring time. BGPC coloring is especially expensive for `Stencil`, even though the resulting color count is almost the same as for D2. Thus, for the CPU backend, cheap coloring is more important than finding a slightly better color count.

The GPU results are more mixed. Dense GPU execution is a strong baseline for some cases, but it can still be beaten by sparse CPU pipelines when the number of compressed directions is small. BGPC GPU is also highly benchmark-dependent. It performs very well on `Banded5` JVP, especially on the largest instance, but it is much weaker on `BA`, `HT`, `Stencil`, and on the VJP benchmarks. This suggests that GPU performance depends not only on the number of colors, but also on which side of the sparsity pattern is colored and how well the resulting computation fits the GPU.

There are also some limitations. The implementation assumes that the sparsity pattern is provided by the user, so the cost of discovering it is not included. The main benchmark results use compressed sparse output, so applications that require CSR output must also account for reconstruction cost. The ADBench comparison also shows that problem-specific `calculate_jacobian` implementations can still be much faster, especially on GPU. This is expected, since those im-

plementations are specialized to the benchmark problems, while our pipeline is designed to work from a user-provided sparsity pattern.

Future work could improve several parts of the pipeline. One motivation for implementing BGPC was to obtain a coloring method that was more suitable for the GPU than the sequential D2 coloring algorithm. The results show that this is partly successful, but the current BGPC implementation is not robust across benchmarks and is not competitive with D2 on CPU. Future work could therefore investigate more GPU-friendly coloring algorithms or a hybrid pipeline where coloring is performed on the CPU and compressed AD evaluation is performed on the GPU. Other directions include automatic sparsity-pattern discovery, more parallel CSR reconstruction, and additional sparse Jacobian methods such as cross-country algorithms.

# 5 Conclusion

This thesis investigated how a known sparsity pattern can be used to compute Jacobians more efficiently in Futhark. The result is a sparse Jacobian library that represents sparsity patterns in CSR form, colors the corresponding bipartite graph, and uses compressed JVPs or VJPs to compute the structurally nonzero Jacobian entries.

The correctness tests validate the implementation layer by layer and compare sparse outputs against dense Jacobians restricted to the known sparsity pattern. The benchmarks show that sparse compression can give large speedups when the number of colors is small compared with the matrix dimension and the reduction in AD work outweighs the coloring overhead. Across the tested problems, D2 CPU is the most robust sparse variant: it is faster than dense CPU in all cases and faster than dense GPU in most cases. BGPC is more backend- and problem-dependent. It performs very well on `Banded5` JVP, especially on the largest instance, but is less consistent overall. The D2/BGPC comparison also shows that, for these benchmarks, the main practical issue is coloring time rather than color count.

The main limitation is that the sparsity pattern must be provided by the user, and problem-specific implementations such as ADBench `calculate_jacobian` can still be much faster, especially on GPU. Future work could investigate automatic sparsity-pattern discovery, more GPU-friendly coloring, hybrid CPU/GPU execution, and additional methods such as cross-country algorithms.

# References


[1] Mathieu Blondel and Vincent Roulet. The Elements of Differentiable Programming. https://arxiv.org/abs/2403.14606, 2024. arXiv:2403.14606, draft version 3, last revised June 24, 2025.

[2] Futhark Developers. Futhark prelude AD documentation. https://futhark-lang.org/docs/prelude/doc/prelude/ad.html, 2026. Accessed 2026-06-03.

[3] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. ColPack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software*, 40(1):1:1–1:31, 2013. doi: 10.1145/2513109.2513110.

[4] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What Color Is Your Jacobian? Graph Coloring for Computing Derivatives. *SIAM Review*, 47(4):629–705, 2005. doi: 10.1137/S0036144504444711.

[5] GradBench Developers. GradBench: A benchmark suite for differentiable programming. https://github.com/gradbench/gradbench, 2026. Accessed 2026-06-02.

[6] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571. ACM, 2017. doi: 10.1145/3062341.3062354.

[7] NVIDIA Corporation. NVPL Sparse: Sparse matrix formats. https://docs.nvidia.com/nvpl/25.1/sparse/storage_format/sparse_matrix.html, 2024. Accessed 2026-06-07.

[8] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. AD for an array language with nested parallelism. https://arxiv.org/abs/2202.10297, 2022. arXiv:2202.10297.

[9] Filip Šrajer, Zuzana Kukelova, and Andrew Fitzgibbon. A Benchmark of Selected Algorithmic Differentiation Tools on Some Problems in Computer Vision and Machine Learning. https://arxiv.org/abs/1807.10129, 2018. arXiv:1807.10129.

[10] Mustafa Kemal Taş, Kamer Kaya, and Erik Saule. Greed is Good: Optimistic Algorithms for Bipartite-Graph Partial Coloring on Multicore Architectures. https://arxiv.org/abs/1701.02628, 2017. arXiv:1701.02628.

# 6 Appendix

## Benchmark Tables

All runtimes are reported in microseconds. CPU denotes the Futhark multicore backend, and GPU denotes the Futhark CUDA backend. Sparse pipeline timings use compressed sparse output unless otherwise stated. Speedups are reported in the main text. Color counts are discussed in the main text.

## Structured Problems

| Case | Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|---|
| Banded5 | $512 \times 16384$ | 15651 | 3465 | 512 | 3540 | 623 |
| Banded5 | $1024 \times 32768$ | 74388 | 14548 | 620 | 4566 | 651 |
| Banded5 | $2048 \times 65536$ | 295952 | 56465 | 1358 | 6183 | 675 |
| Stencil | $4096 \times 4096$ | 23377 | 1357 | 701 | 35367 | 24890 |
| Stencil | $9216 \times 9216$ | 86072 | 6755 | 843 | 251608 | 100974 |
| Stencil | $16384 \times 16384$ | 260597 | 23571 | 1250 | 613216 | 182284 |

Table 11: Structured full JVP pipeline benchmark. Times are in microseconds.

| Case | Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|---|
| Banded5 | $2048 \times 4096$ | 1127 | 422 | 452 | 27957 | 74379 |
| Banded5 | $4096 \times 8192$ | 3891 | 1560 | 558 | 138558 | 153030 |
| Banded5 | $8192 \times 16384$ | 18022 | 6565 | 685 | 516952 | 312551 |
| Stencil | $4096 \times 4096$ | 3721 | 2077 | 1966 | 40586 | 24596 |
| Stencil | $9216 \times 9216$ | 18079 | 11404 | 3207 | 249262 | 100357 |
| Stencil | $16384 \times 16384$ | 57130 | 39595 | 1255 | 605593 | 182193 |

Table 12: Structured full VJP pipeline benchmark. Times are in microseconds.

| Case | Jacobian size | D2 full CPU | D2 color CPU | D2 precolored CPU | BGPC color CPU |
|---|---|---|---|---|---|
| Banded5 | $512 \times 16384$ | 603 | 40 | 475 | 706 |
| Banded5 | $1024 \times 32768$ | 731 | 185 | 453 | 3091 |
| Banded5 | $2048 \times 65536$ | 1085 | 405 | 467 | 4301 |
| Stencil | $4096 \times 4096$ | 553 | 128 | 416 | 29133 |
| Stencil | $9216 \times 9216$ | 746 | 278 | 422 | 220449 |
| Stencil | $16384 \times 16384$ | 972 | 505 | 444 | 654910 |

Table 13: Structured pipeline breakdown benchmark. Times are in microseconds. The precolored column gives the D2 CPU pipeline time when colors are supplied as input.

## Bundle Adjustment

| Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|
| $6144 \times 2784$ | 20566 | 900 | 1350 | 8327 | 14986 |
| $12288 \times 5200$ | 54673 | 3105 | 2449 | 11830 | 22648 |
| $24576 \times 9664$ | 118479 | 10910 | 5438 | 21846 | 36458 |
| $36864 \times 14128$ | 210175 | 25133 | 7139 | 28142 | 51129 |

Table 14: `BA` full JVP pipeline benchmark. Times are in microseconds.

| Jacobian size | D2 full CPU | D2 color CPU | D2 precolored CPU | BGPC color CPU |
|---|---|---|---|---|
| 6144 × 2784 | 1456 | 1047 | 429 | 7297 |
| 12288 × 5200 | 2919 | 2105 | 432 | 10621 |
| 24576 × 9664 | 5011 | 4285 | 1105 | 17925 |
| 36864 × 14128 | 6831 | 6261 | 1562 | 25563 |

Table 15: `BA` pipeline breakdown benchmark. Times are in microseconds. The precolored column gives the D2 CPU pipeline time when colors are supplied as input.

### Hand Tracking

| Jacobian size | Dense CPU | Dense GPU | D2 CPU | BGPC CPU | BGPC GPU |
|---|---|---|---|---|---|
| 1536 × 1050 | 17756 | 23921 | 12265 | 26842 | 440231 |
| 3072 × 2074 | 59801 | 90906 | 20202 | 50192 | 884424 |
| 6144 × 4122 | 242386 | 353114 | 35476 | 98151 | 1771523 |
| 12288 × 8218 | 903892 | 1394966 | 59358 | 193520 | 3545961 |

Table 16: `HT` full JVP pipeline benchmark. Times are in microseconds.

| Jacobian size | D2 full CPU | D2 color CPU | D2 precolored CPU | BGPC color CPU |
|---|---|---|---|---|
| 1536 × 1050 | 12983 | 1062 | 8620 | 16063 |
| 3072 × 2074 | 21587 | 2138 | 12249 | 34547 |
| 6144 × 4122 | 30408 | 4277 | 23719 | 70228 |
| 12288 × 8218 | 53996 | 8525 | 40801 | 135152 |

Table 17: `HT` pipeline breakdown benchmark. Times are in microseconds. The precolored column gives the D2 CPU pipeline time when colors are supplied as input.

### ADBench `calculate_jacobian` Comparisons

| Jacobian size | D2 compressed CPU | D2 CSR CPU | ADBench CPU | ADBench GPU |
|---|---|---|---|---|
| 6144 × 2784 | 1426 | 2385 | 375 | 95 |
| 12288 × 5200 | 3092 | 3949 | 945 | 97 |
| 24576 × 9664 | 5017 | 7204 | 3237 | 100 |
| 36864 × 14128 | 7132 | 11024 | 4285 | 103 |

Table 18: `BA` ADBench comparison. Times are in microseconds.

| Jacobian size | D2 compressed CPU | D2 CSR CPU | ADBench CPU | ADBench GPU |
|---|---|---|---|---|
| 1536 × 1050 | 13360 | 11223 | 17153 | 882 |
| 3072 × 2074 | 17270 | 19123 | 17558 | 1425 |
| 6144 × 4122 | 29478 | 29324 | 27427 | 2523 |
| 12288 × 8218 | 54057 | 56638 | 48335 | 5103 |

Table 19: `HT` ADBench comparison. Times are in microseconds.