Benchmarking Futhark-AD using MINPACK-2

Project Outside Course Scope, 7.5 ECTS

Peter Kanstrup Larsen
Supervisor: Cosmin Eugen Oancea

January 2023

1 Abstract

The purpose of this project has been twofold. First, to review literature for techniques implementing automatic differentiation (AD). Second, to identify and implement a set of benchmarks in Futhark that exhibit sparse Jacobian/Hessian matrices.
To do this, we have translated a problem from the FORTRAN MINPACK-2 collection and benchmarked the Futhark-AD on this. This gave promising results when looking at speedup, but the generated Jacobians do not validate. Furhter investigation of the sparsity patterns have not been made due to time limitations.

The project code can be found in this GitHub repository:

`https://github.com/PeterLarsen404/minpack2-fut`

2 Introduction

Computing the derivatives of an objective function in computer programs is a fundamental operation in several fields such as optimization, neural networks, and physics-based modeling. In optimization, differentiation is used to compute gradients of a function, which are commonly used in optimization algorithms such as gradient descent. In neural networks, gradients of a model's parameters with respect to a loss function are used in the network's training and inference.

The methods used for computing the derivatives are classified into four categories: Manual differentiation, numerical differentiation, symbolic differentiation, and automatic differentiation (AD):

2.0.1 Manual differentiation

Manual differentiation calculates a function's derivative by using the calculus rules. As the name implies, it is done by hand rather than using a computer program or calculator. The process involves applying rules, such as the power rule, the product rule, and the chain rule. This method is mainly used for simple functions since it can become tedious and time-consuming for more complex, or multiple-variable functions.

2.0.2 Numerical differentation

Numerical differentiation is a method of approximating the derivative of a function using numerical techniques. It is often applied when it is difficult to obtain the analytical form of the derivative. The basic idea is to approximate the derivative by measuring the change in the function's value for small changes in the independent variable.

However, the accuracy of numerical differentiation methods depends on the step size used, which sometimes leads to less accurate results. Additionally, numerical differentiation methods can have numerical errors, such as round-off errors, and truncation errors.[1]

2.0.3 Symbolic differentiation

Symbolic differentiation involves manipulating the function using the rules of calculus and algebra to obtain an expression for the derivative. The result of symbolic differentiation is an exact expression for the derivative, represented by mathematical symbols rather than numerical values. A drawback of using symbolic derivatives is that the derivatives grow exponentially, resulting in unmaintainable code [6], and can lead to inefficient runtime when evaluated [3].

2.0.4 Automatic differentiation

Lastly, we have automatic differentiation which will be the main method of this paper. The idea behind AD is that the process of differentiation is performed using a computer program. We will dive deeper into the exact computation in the next section.

Examples of the different differentiation methods can be observed in Figure 1.

[1] https://en.wikipedia.org/wiki/Numerical_differentiation

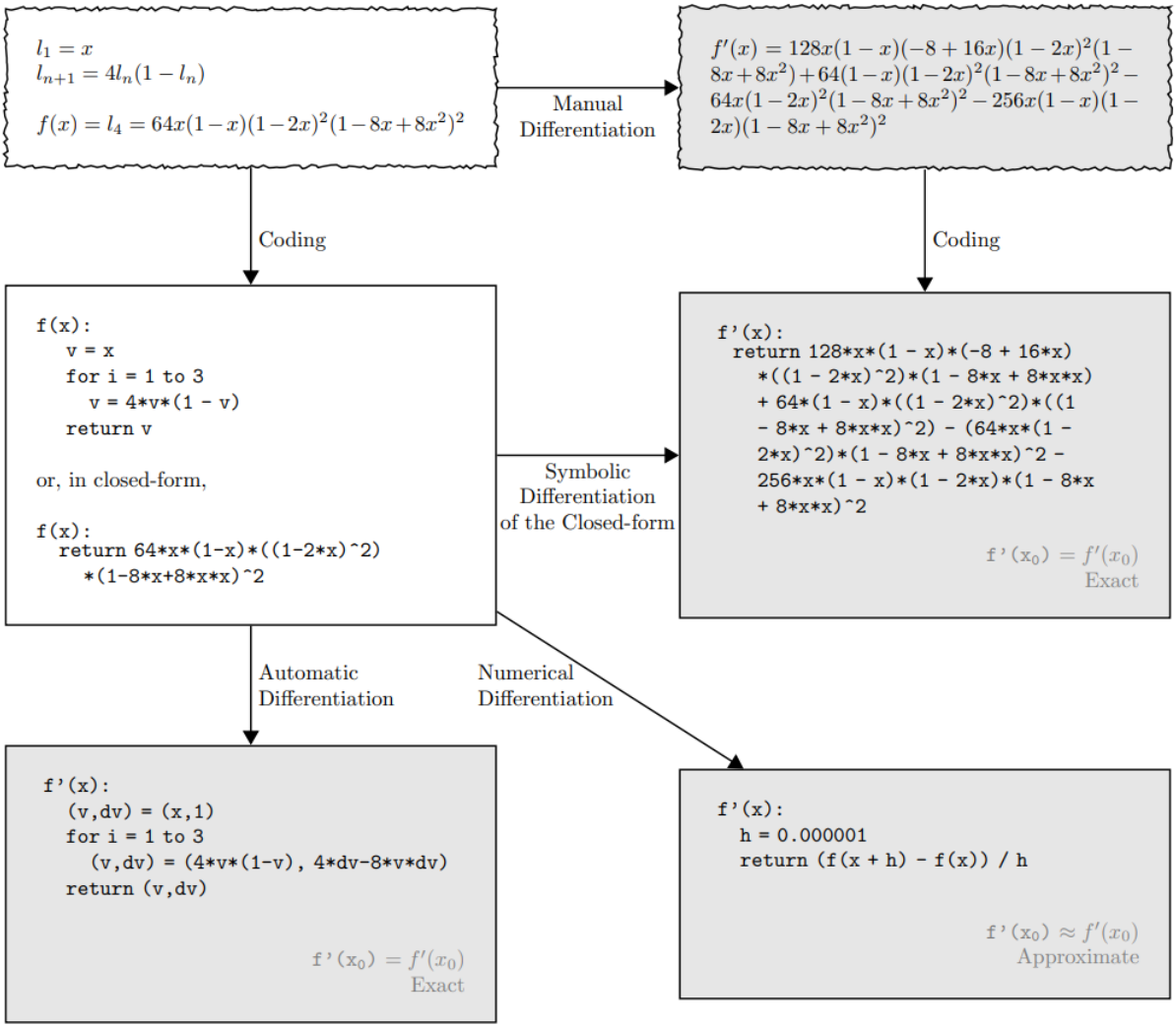

Figure 1: Examples of differentiation methods (image from [2]).

Futhark-AD is an AD library made to compute the derivatives of Futhark programs. Futhark is a statically typed, data-parallel, and purely functional array language designed to be compiled to efficient parallel code [5].

In this paper, we look into how well the Futhark-AD library handles the computation of Jacobian matrices. To do this, we use a problem from the MINPACK-2 test collection and investigate the validity and runtime of the computation.

3 Automatic differentiation

AD exploits that every function, no matter how complicated, is executed on a computer as a sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and log. By successively applying the chain rules on the composition of operations, the exact derivatives can be computed in a mechanical fashion.

There are two basic modes of AD, which are usually referred to as *forward mode* and *reverse mode*. The following notation and object function are used in the following sections:
A function $f : \mathbb{R}^n \to \mathbb{R}^m$ consists of the intermediate variables v_i with:

1. $v_{i-n} = x_i,\ i = 1, ..., n$ Are the input variables.
2. $v_i,\ i = 1, ..., l$ Are the intermediate variables.
3. $y_{m-i} = v_{l-i},\ i = m-1, ..., 0$ Are the output variables.
4. $\dot{v} = \frac{\partial v}{\partial x}$ Is the derivative of the variable.

We let $y = f(x_1, x_2) = x_1^2 - x_1 \cdot cos(x_2) + sin(x_1)$, which gives us the following computational graph:

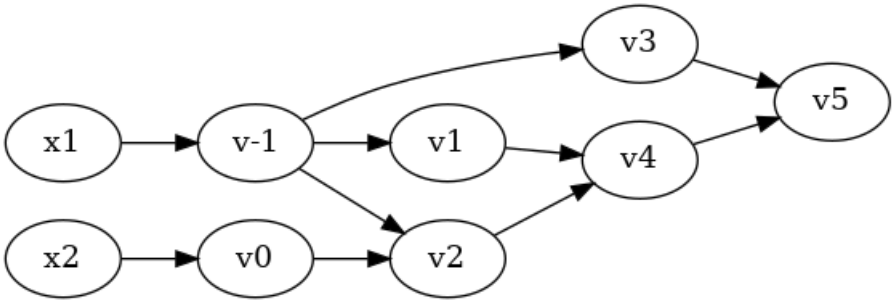

Figure 2: Computational graph of the example: $y = f(x_1, x_2) = x_1^2 - x_1 \cdot cos(x_2) + sin(x_1)$

3.1 Forward mode

To compute the derivative using the forward mode, we apply the chain rule to each operation in the forward primal trace. We evaluate each primal in lockstep with its corresponding derivative and obtain the derivative of the function in the final variable of the derivative trace [2].

3.1.1 Forward mode AD example

Let $y = f(x_1, x_2) = x_1^2 - x_1 \cdot cos(x_2) + sin(x_1)$. We want to evaluate it at $(x_1, x_2) = (3, 8)$.
Forward Primal Trace:

$$
\begin{aligned}
v_{-1} &= x_1 && = 3 && (1)\\
v_0 &= x_2 && = 8 && (2)\\
v_1 &= v_{-1}^2 && = 3^2 && (3)\\
v_2 &= v_{-1} \cdot cos(v_0) && = 3 \cdot cos(8) && (4)\\
v_3 &= sin(v_{-1}) && = sin(3) && (5)\\
v_4 &= v_1 - v_2 && = 9 + 0.4365 && (6)\\
v_5 &= v_4 + v_3 && = 9.4365 + 0.1411 && (7)\\
y &= v_5 && = 9.5776 && (8)
\end{aligned}
$$

For the Forward Tangent Trace, we let $\dot{x} = 1$ to compute $\frac{\partial y}{\partial x}$:

$$
\begin{aligned}
\dot{v}_{-1} &= \dot{x_1} && = 1 && (9)\\
\dot{v}_0 &= \dot{x_2} && = 0 && (10)\\
\dot{v}_1 &= 2 \cdot v_{-1} && = 2 \cdot 3 && (11)\\
\dot{v}_2 &= \dot{v}_{-1} \cdot cos(v_0) + \dot{v}_0 \cdot v_{-1} && = 1 * cos(8) + 0 * 3 && (12)\\
\dot{v}_3 &= cos(v_{-1}) && = cos(3) && (13)\\
\dot{v}_4 &= \dot{v}_1 - \dot{v}_2 && = 6 + 0.1455 && (14)\\
\dot{v}_5 &= \dot{v}_4 + \dot{v}_3 && = 6.1455 - 0.99 && (15)\\
\dot{y} &= \dot{v}_5 && = 5.1555 && (16)
\end{aligned}
$$

Observing the above example we see that the forward mode can compute the Jacobian matrix one column each evaluation, this is done by initializing one of the variables to be 1 and the rest to zero. Thus, it takes n evaluations to compute the full Jacobian.

Instead of using unit vectors to compute the entire Jacobian, we can use an arbitrary vector r to easily compute the Jacobian-vector product $\nabla f * r$ in only only one evaluation.

The run time for computing the Jacobian using the forward mode is: $n \cdot c \cdot ops(f)$, where c is a constant guaranteed to be $c < 6$ [7], and $ops(f)$ is the number of operations required to evaluate f.

Forward AD is very advantageous when working with functions of type $f : \mathbb{R} \to \mathbb{R}^m$ because every derivative can be computed in only one evaluation. The method does however lack when the function is of the type $f : \mathbb{R}^n \to \mathbb{R}$.

3.1.2 Forward-AD in Futhark

The implementation of forward mode AD in Futhark [9] is closely related to the "dual number formulation" described in [2]. The use of dual numbers in forward mode AD is very helpful because they have the property of automatically calculating the derivative of a function while calculating the value of the function.

We denote dual numbers with ϵ and give them the property of being a nilpotent number such that $\epsilon^2 = 0$ and $\epsilon \neq 0$. The behavior of dual numbers is somewhat similar to complex numbers when using math operators.[2]

Simple Example of dual numbers:[a]
Let $f(x) = 3x + 2$. We want to evaluate it at 4:
Convert 4 to dual number:

$$4 + 1\epsilon \tag{17}$$

Multiply by 3:

$$(4 + 1\epsilon) \cdot (3 + 0\epsilon) = 12 + 3\epsilon \tag{18}$$

Add 2:

$$14 + 3\epsilon \tag{19}$$

Thus:

$$f(4) = 14 \quad \text{and} \quad f'(4) = 3 \tag{20}$$

[a] Example taken from: https://blog.demofox.org/2014/12/30/dual-numbers-automatic-differentiation/

In Futhark-AD, the higher-order function corresponding to the forward mode is `jvp`, with the following types:

$\texttt{jvp} : (f : \alpha \to \beta) \to (x : \alpha) \to (dx : \alpha) \to \beta$

Where: `f` is the function we want to find the derivative of. `x` is the point of evaluation. `dx` is the input tangents.
jvp stands for "Jacobian-vector product" and computes: $\mathbf{J}(f(x)) \cdot dx$

[2] https://en.wikipedia.org/wiki/Dual_number

3.2 Reverse mode

The reverse mode of AD is also known as backpropagation. The idea is first to evaluate the function at a specific input, and then propagate the gradient of the output with respect to the input backward through the computation, applying the chain rule at each operation - The contrast to forward mode AD.

We introduce a new notation: $\bar{v_i} = \frac{\partial y_j}{\partial v_i}$ denoting the contribution of the change in each variable v_i to the change in the output y.

3.2.1 Reverse mode AD example

Let $y = f(x_1, x_2) = x_1^2 - x_1 \cdot cos(x_2) + sin(x_1)$. We want to evaluate it at $(x_1, x_2) = (3, 8)$.
We start with the Forward Primal Trace, which will be the same as the forward mode:

$$v_{-1} = x_1 = 3 \tag{21}$$
$$v_0 = x_2 = 8 \tag{22}$$
$$v_1 = v_{-1}^2 = 3^2 \tag{23}$$
$$v_2 = v_{-1} \cdot cos(v_0) = 3 \cdot cos(8) \tag{24}$$
$$v_3 = sin(v_{-1}) = sin(3) \tag{25}$$
$$v_4 = v_1 - v_2 = 9 + 0.4365 \tag{26}$$
$$v_5 = v_4 + v_3 = 9.4365 + 0.1411 \tag{27}$$
$$y = v_5 = 9.5776 \tag{28}$$

For the Reverse Adjoint Trace, we let $\bar{v_5} = \bar{y} = \frac{\partial y}{\partial y} = 1$ to compute both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$. Recall that this is computed "bottom-up":

$$\bar{v}_5 = \bar{y} = 1$$
$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \cdot 1 = 1$$
$$\bar{v_3} = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \cdot 1 = 1$$
$$\bar{v_1} = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \cdot 1 = 1$$
$$\bar{v_2} = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_4 \cdot (-1) = -1$$
$$\bar{v}_{-1} = \bar{v}_3 \frac{\partial v_3}{\partial v_{-1}} = \bar{v}_3 \cdot cos(v_{-1}) = -0.99$$
$$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \bar{v}_{-1} + \bar{v}_2 \cdot cos(v_0) = -0.844$$
$$\bar{v_0} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \bar{v}_2 \cdot v_{-1} \cdot (-sin(v_0)) = -2.968$$
$$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = \bar{v}_{-1} + \bar{v}_1 \cdot 2 \cdot v_{-1} = 5.156$$
$$\bar{x}_1 = \bar{v}_{-1} = 5.156$$
$$\bar{x}_2 = \bar{v}_0 = 2.968$$

Compared to the forward mode, the reverse mode is significantly less costly to evaluate (in terms of operation count), when the functions have a large number

of inputs. Looking at the example when the function is of the type $f : \mathbb{R}^n \to \mathbb{R}$, reverse mode only needs one evaluation compared to the n evaluations needed by the forward mode.

This trait is very advantageous from a machine learning point of view since the objective often includes many parameters, and the gradient of this objective is used in the backpropagation of the model. However, since we use the operations of the evaluated function in the reverse adjoint trace, we need to store them. As the number of operations increases, the amount of storage required also increases proportionally, which may cause issues [4].

The runtime for computing the Jacobian using the reverse mode is: $m \cdot c \cdot ops(f)$.

3.3 Reverse-AD in Futhark

In reverse mode, intermediate program states are needed to compute the reverse adjoint trace. This is however not an easy task to solve from a parallel execution point of view. Normally the variables are stored on a *tape* and abstractions such as checkpointing [4] are used to efficiently make the data transfer. However, these approaches are not suited for parallel context and are difficult to implement when concerning spatial and temporal locality across scopes [9].

To solve this, authors of [9] requires the return sweep to recompute the forward sweep in order to bring all the needed variables into scope. This solution solves the problem as it does not modify the work-span asymptotic since the recomputation overhead is at worst proportional to the depth of the deepest nest of scopes, which is constant for a given non-recursive program.

Additionally, reverse AD replaces reads with accumulations, which are not suited for the data-parallel constructs used in Futhark. The authors of [9] solves this by describing a set of rewrite rules, and introduce *accumulators* to handle free variables.

In Futhark-AD, the higher-order function corresponding to the forward mode is `vjp`, with the following types:

$\texttt{vjp} : (f : \alpha \to \beta) \to (x : \alpha) \to (dy : \beta) \to \alpha$

Where: `f` is the function we want to find the derivative of. `x` is the point of evaluation. `dy` is the output adjoints.
vjp stands for "vector-Jacobian product" and computes: $dy^T \cdot \mathbf{J}(f(x))$

4 MINPACK-2-fut

To test Futhark-AD we take inspiration from the MINPACK-2 test problem collection.

MINPACK-2 is a FORTRAN library of subroutines for solving systems of nonlinear equations, least squares problems, and minimization problems. The problems in the collection comes from a real application and are chosen with the following requirement in mind: *Each problem must come from a real application and be representative of other commonly encountered problems.* Thus, the collection includes problems from many fields such as: Fluid dynamics, medicine, combustion, superconductivity, and more [1].

4.1 Flow in a Channel problem

We have chosen to test Futhark-AD on the "Flow in a Channel" problem from MINPACK-2, which deals with the boundary value problem that fluid injection through one side of a long vertical channel introduces.

The FORTRAN subroutine `dficfj` found in the `dficfj.f`[3] file, corresponds to this problem. The subroutine takes 8 arguments, of which only the following are interesting for this paper:

`x`: Vector containing starting point.

`r`: Reynolds number[4]

`nint`: Number of subintervals in the collocation method used to discretize the boundary value problem (Is always: $\frac{\texttt{size of x}}{8}$)

`task`: Determines the behavior of `dficfj` depending on the character:

- `XS`: Return the solution of the flow in a channel problem where Reynolds number is 0. This is the standard starting point, and can be used as `x`.
- `F`: Evaluate the function at `x`.
- `J`: Evaluate the Jacobian matrix at `x`.

[3] FORTRAN code found at: https://ftp.mcs.anl.gov/pub/MINPACK-2/tprobs/dficfj.f
[4] https://en.wikipedia.org/wiki/Reynolds_number

4.1.1 Translation and design

To test Futhark-AD, a translation from the FORTRAN subroutine to a Futhark function was required. Since Futhark-AD computes the Jacobian of the function, we focused on translating the parts corresponding to task `F` and `XS`. The following describes the implemented Futhark functions. The code is found in the appendix section:

task_XS (nint : i64) : [nint*8]f64
This function corresponds to the work made when `task = 'XS'`. The argument `nint` is used to determine the size of the output array, which is equal to `nint*8`. We use this function to generate the input vectors `x` needed for `dficfj`.

mk_eq (nint_ : i64) (r: f64) (b : i64) (rhnfhk : [4][8][8][5]f64) (x : [nint*8]f64) : [8]f64
This helper function is used by the objective function `dficfj` to compute the collocation and continuity equations. The function is intended to be used in the following way: `tabulate nint (\i -> mk_eq (nint-1) r i rhnfhk x)`. We compute 8 equations for each evaluation of `mk_eq`.

We use the following arguments:

`nint_`: Should always be `nint - 1` since it determines when the last 4 continuity equations are computed.

`r`: Reynolds number.

`b`: Used to keep track of how many equations has been made in total.

`rhnfhk`: Array which stores every possible combination of `rho`, `h`, and n factorial. (This is computed by the function `dficfj`).

`x`: Array determining the evaluation point of the objective function. (Also the array generated by `task_XS`).

dficfj (nint : i64) (x : [nint*8]f64) (r : f64) : [nint*8]f64
This is the objective function and corresponds to the work made when `task = 'F'`.

We firstly use the predefined `rho` values to compute the $4 \times 8 \times 8 \times 5$ array `rhoijhs`, which stores every possible combination of `rho`, `h`, and n factorial. The values of `rhoijhs` only changes depending on `h`, which is equal to $\frac{1}{\texttt{nint}}$

We then use `mk_eq` to compute the `fvec` array, which contains the function evaluated at `x`, and return it.

We use the following arguments:

`nint`: Number of subintervals.

`x`: Array determining the evaluation point. (Could be the array generated by `task_XS`).

`r`: Reynolds number.

4.1.2 Implementation

The primary focus of the translation has been to obtain as much parallelism as possible using the parallel constructs available in Futhark. Examples of this are, including, but not limited to:

Generation of x
Originally, the computation of `x` when `task = 'XS'` was done using `nint` loop iterations. However, since we have no dependencies between each iteration we could potentially do every iteration in parallel. Using the keyword `loop` in Futhark results in an entirely sequential execution [5], so we opted to use `tabulate nint`, generating 8 elements of x, *nint* times and flatten the array.

Generation of rhnfhk
Originally, `rhnfhk` was computed using a nest of 4 loops with no dependencies. Hence, we could use a nest of `tabulate` and `tabulate_3d` to compute this in parallel. The "helper" arrays: `hms`, `rhnfhk`, `nfs` contain the values needed for the final calculation and are easily iterated through by the nested tabulates.

4.1.3 Usage of Futhark-AD

In this paper, we aim to investigate the computation of the Jacobian of `dficfj` with respect to `x`, hence the Jacobian would be a matrix of size: $(\texttt{nint} * 8) \times (\texttt{nint} * 8)$.

We therefore make the function `dficfj_test_ad`, which is just a rewritten `dficfj` in order to only use `x` as the input argument. We use 1 as Reynolds number.

```
1 def dficfj_test_ad [n] (x : [n]f64) : [n]f64 =
2   let nint = n/8i64
3   in dficfj nint x 1f64 :> [n]f64
```

To compute the Jacobians, we apply the `jvp` and `vjp` functions from Futhark-AD on `dficfj_test_ad`. Since the functions only compute one coloum/row for each evaluation, we use them in a `tabulate` to compute the entire Jacobian matrix:

```
1 forward_ad_jac = tabulate n (\ i ->
2         jvp dficfj_test_ad x (replicate n 0 with [i] = 1))
3
4 reverse_ad_jac = transpose (tabulate n (\ i ->
5         vjp dficfj_test_ad x (replicate n 0 with [i] = 1)))
```

Where `n` is equal to the length of `x`. Note that reverse_ad_jac is transposed.

5 Benchmarks and validation

The input datasets used for benchmarking and testing is generated using `task_XS`, and `nint` values from 1 to 1000000.

5.1 Benchmarks

To benchmark Futhark-AD and our implementation of `dficfj`, we take the average runtime of the Futhark functions using the `futhark bench` command.

Since we want to know how well the implementations compare to the original FORTRAN subroutines, we compile these to `c` and time the functions using the `clock` function. All benchmarks have been run on an A-100 GPU and an Intel 8-core 1.60GHz CPU:

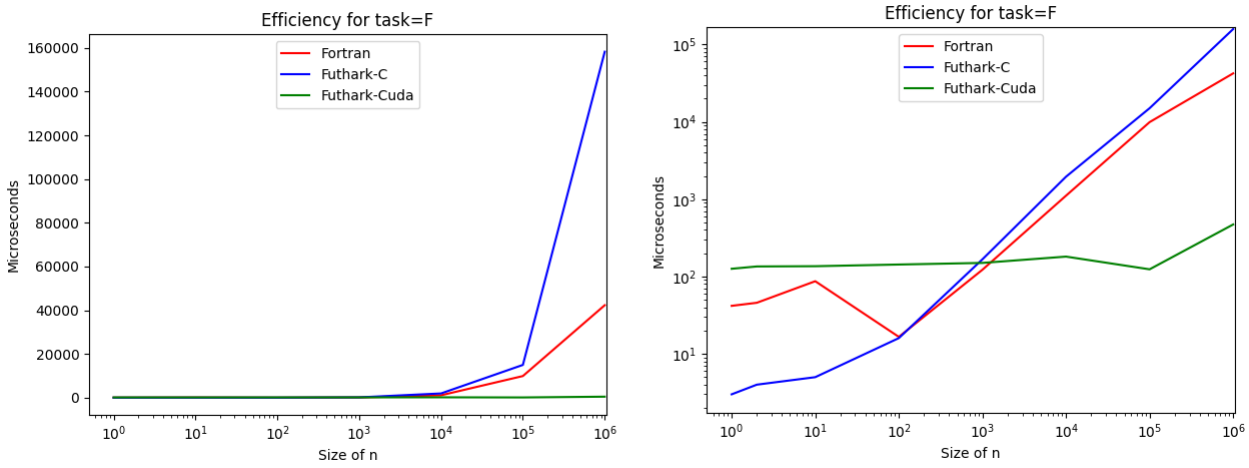

Figure 3: Plot showing the runtime for the functions corresponding to when `task = 'F'`. Left plot shows microseconds on a linear scale, right on a logarithmic.

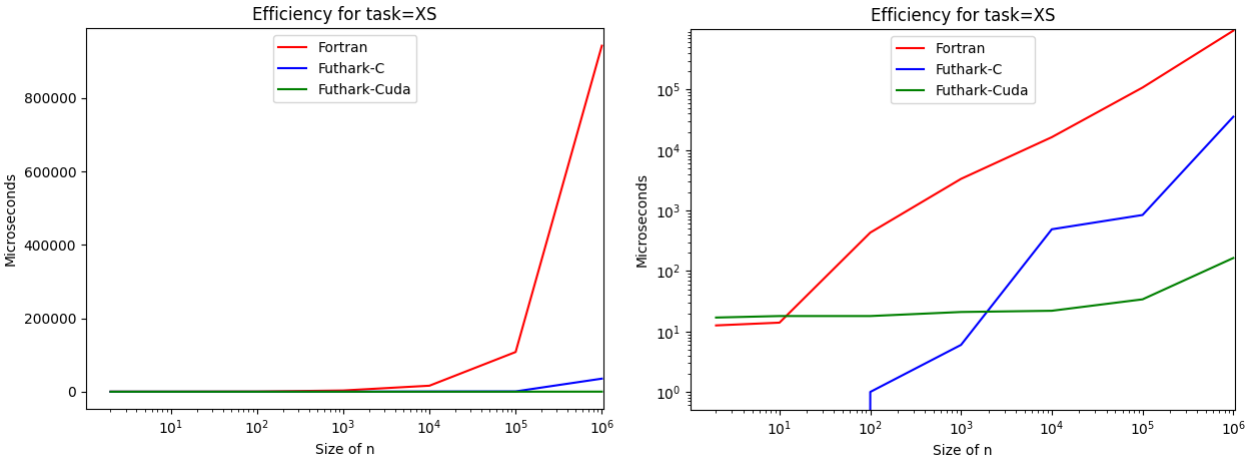

Figure 4: Plot showing the runtime for the functions corresponding to when `task = 'XS'`. Left plot shows microseconds on a linear scale, right on a logarithmic.

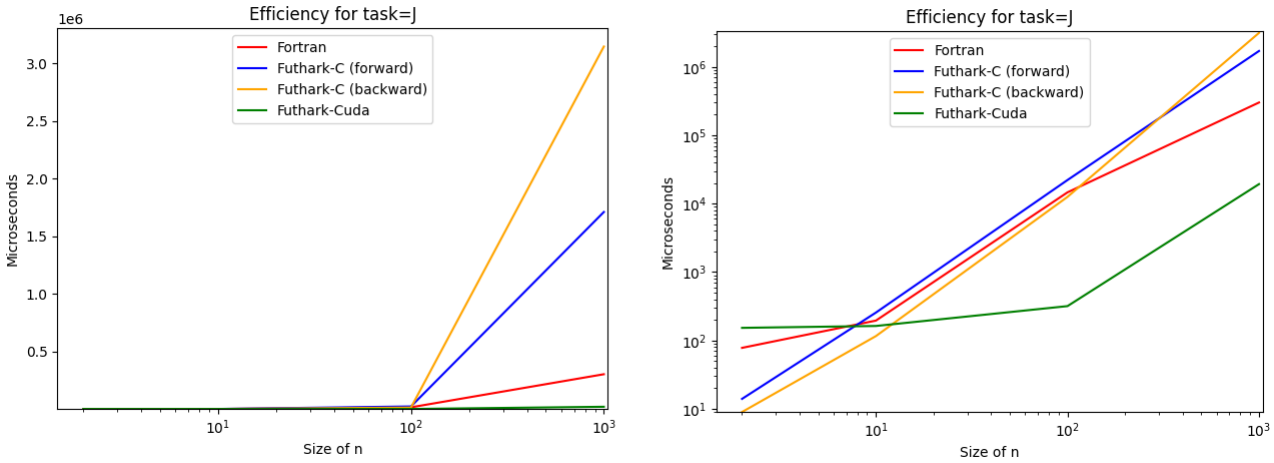

Figure 5: Plot showing the runtime for the functions corresponding to when `task = 'J'`. Left plot shows microseconds on a linear scale, right on a logarithmic.

As expected, `dficfj` and `task_XS` are very fast for large values when executed on a GPU. This is because we have no dependencies between loops and can perform them in parallel. Most interesting is Figure: 5 since it compares the computation of Jacobians in FORTRAN with Futhark-AD. We observe promising results for the runtime of forward mode in Futhark-AD, outperforming FORTRAN for values larger than 10^1.

5.2 Validation

To test the validity of our implementation `dficfj`, and the Jacobian generated by Futhark-AD, we compare the output of the functions with the output computed by the original FORTRAN subroutines using `futhark test`:

`Dficfj:` Validates.

`TaskXS:` Validates.

`jvp/vjp:` Fails.

When `nint` $= 1$, the 8×8 Jacobian computed by `jvp` has 5 mismatches, the use of `vjp` results in 31 mismatches.

Looking at the mismatches, we see that the elements from `jvp` is off by a small margin. More interesting is the mismatches from `vjp`: 4 out of 8 columns consists entirely of 0.

Moreover, using "cuda" as backend for compiling the reverse mode Futhark-AD, results in a compiler error.

Further investigation of the computed Jacobian and the Futhark compiler is needed to conclude anything from these results.

6 Discussion and Furture work

Naturally, investigation of the validity should be the first priority moving forward. Benchmarks for functions doing the same work but yielding different results are challenging and mostly uninteresting. However, if the errors only occur in special cases and can be avoided using different translation strategies, our results might hint to the efficiency of performing AD on the GPU.

The MINPACK-2 test collection includes far more problems, with real-world use cases. Further translation of the collection would not only bring new interesting problems to solve, but would also test the robustness and runtime of Futhark-AD more thoroughly.

When potential compiler errors have been solved, the next step would be to further investigate how matrix sparsity affects the performance of Futhark-AD. Additionally, the work on array short circuiting as shown in [8] might improve the speedup even further, particularly for array initialization where certain slices have predetermined values (such as in `x`, which has intervals of 4 zeroes)

7 Conclusion

This project's objective was to explore the Futhark-AD tool using the FORTRAN MINPACK-2 test collection. To do this, we successfully translated the "Flow in a Channel problem" to Futhark, focusing on maximizing the parallelism. We then benchmarked the forward and reverse modes of Futhark-AD using the translated subproblem as the objective function.

The results were promising from a runtime point of view, achieving a big speedup when executed on a GPU. Unfortunately, the computed Jacobians do not validate when compared to the Jacobians computed by the FORTRAN subroutine. This issue remains to be solved in future work.

References

[1] Brett M Averick, Richard G Carter, Guo-Liang Xue, and JJ Moré. The minpack-2 test problem collection. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 1992.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

[3] George F Corliss. Applications of differentiation arithmetic. In *Reliability in computing*, pages 127–148. Elsevier, 1988.

[4] Benjamin Dauvergne and Laurent Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *Computational Science–ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV 6*, pages 566–573. Springer, 2006.

[5] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2018.

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

[7] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[8] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. Memory optimizations in an array language. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 424–438, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[9] R. Schenck, O. Ronning, T. Henriksen, and C. E. Oancea. Ad for an array language with nested parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 829–843, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

8 Code

8.1 task_XS

```
def task_XS (nint : i64) : []f64 =
  let zero   : f64 = 0.0
  let one    : f64 = 1.0
  let two    : f64 = 2.0
  let three  : f64 = 3.0
  let six    : f64 = 6.0
  let twelve : f64 = 12.0

  let h : f64 = 1f64 / (f64.i64 nint)

  -- Compute 8 elements of x, nint times
  let x = flatten (tabulate nint (\i ->
    let xt = f64.i64 i*h
    in [xt*xt*(three-two*xt),
        six*xt*(one-xt),
        six*(one-two*xt),
        -twelve,zero,zero,zero,zero]))
  in x
```

8.2 mk_eq

```
-- Computes the collocation and continuity equations
-- (4 collocation, 4 continuity).
-- "b" goes from 1 to nint.
def mk_eq [n] (nint_ : i64) (r: f64) (b : i64)
              (rhnfhk : [][][][]f64)
              (x : [n]f64) : [8]f64 =
  let cpts = 4i64
  let deg = 4i64
  let npi = cpts + deg
  let var = b*npi

  -- Create the values for each k in the
  -- collocation equations.
  let col_w_pre_red : [4][5][4]f64 =
    tabulate_3d 4 5 4 (\ k i j ->
      if (j > (4-i)) then
        rhnfhk[k,4+j-i,4+j-i,4-i]*x[var+4+j]
      else
        (rhnfhk[k,j,j,j]*x[var+i+j])+
        (rhnfhk[k,4+j-i,4+j-i,4-i]*x[var+4+j]))

  -- Reduce the array to get w[deg+1] values.
  let col_w_red : [4][5]f64 =
    tabulate_2d 4 5 (\ i j ->
      reduce (+) 0 col_w_pre_red[i,j])

  -- Compute the collocation equations.
  let col_eq : [4]f64 =map (\ w -> w[4] -
    r*(w[1]*w[2]-w[0]*w[3])) col_w_red

  -- Create the values in the collocation equations.
  let con_w_pre_red : [4][4]f64 =
    tabulate_2d 4 4 (\ i j ->
      if (j > (4-i)) then
        rhnfhk[1,0,4+j-i,4-i]*x[var+4+j]
      else
        (rhnfhk[1,0,j,j]*x[var+i+j])+
        (rhnfhk[1,0,4+j-i,4-i]*x[var+4+j]))

  -- Reduce the array to get w[deg] values.
  let con_w_red : [4]f64 =
    map (\ e -> reduce (+) 0 e) con_w_pre_red

  -- Compute the continuity equations.
  -- If i is equal to (nint-1) compute the
  -- last two elements instead of four.
  let con_eq : [4]f64 =
    if (b < nint_) then
```

```
      tabulate 4 (\i -> x[var+cpts+deg+i]-con_w_red[i])
    else
      [con_w_red[0]-1f64,con_w_red[1],0f64,0f64]

-- Concatenate the equations.
in tabulate 8 (\i ->
  if i < 4 then col_eq[i] else con_eq[i-4]) :> [8]f64
```

9 dficfj

```
def dficfj (nint : i64) (x : []f64) (r : f64) : []f64 =
  let cpts : i64 = 4
  let deg  : i64 = 4
  let npi : i64 = cpts + deg
  let dim : i64 = deg + cpts - 1
  let dim_ : i64 = dim+1
  let deg_ : i64 = deg+1

  let h = 1f64 / (f64.i64 nint)

  -- Predefined rho array
  let rho : [4]f64 = [
    0.0694318413734436035,
    0.330009490251541138,
    0.669990539550781250,
    0.930568158626556396
  ]

  -- Store every value of hm in a an array.
  let hms = tabulate deg_ (\i -> h**(f64.i64 i))

  -- Store every value of rhoijh in a 3d array.
  let rhoijhs : [deg_][cpts][dim_]f64 =
    tabulate_3d deg_ cpts dim_ (\ i j k ->
        hms[i]*(rho[j]**(f64.i64 k)))

  -- Store every value of nf in an array.
  let nfs : [dim_]f64 = [1,1,2,2*3,2*3*4,2*3*4*5,2*3*4*5*6
    ,2*3*4*5*6*7] :> [dim_]f64

  -- Create the 4d rhnfhk array:
  let rhnfhk : [deg_][cpts][dim_][dim_]f64 =
    tabulate deg_ (\m ->
      tabulate_3d cpts dim_ dim_ (\ i j k ->
        rhoijhs[m,i,j]/nfs[k]))

  -- Rearrange rhnfhk to make it fit the one in fortran.
  let rhnfhk : [cpts][dim_][dim_][deg_]f64 =
    tabulate cpts (\i ->
      tabulate_3d dim_ dim_ deg_ (\ j k m ->
        rhnfhk[m,i,j,k]))

  -- Compute fvec
  let fvec =
    flatten (map (\ i ->
      mk_eq (nint-1) r i rhnfhk x) (iota nint))

```

```
48    -- Prepend x[0], x[1] and remove last two elements.
49    let fvec = tabulate (npi*nint) (\ i ->
50      if i < 2 then x[i] else fvec[i-2])
51
52    in fvec
```