DPP project

Peter Adema
Sophus Valentin Willumsgaard

January 2025

Contents

1 Introduction

The purpose of this project is to implement the algorithm given in [Pas+21], for reverse differentiation of the scan operator in Futhark. In this report, we describe the procedure for reverse differentiation of a scan operation given in [Pas+21], and how it differs from the current implementation as described in [Bru+24].

We then show how we have implemented the procedure in the Futhark compiler, and lastly give benchmarks to compare with the current implementation.

2 Description of Algorithm

We here give a description of the general-case procedures for automatic differentiation of scan described in [Pas+21] and [Bru+24].

We will use $\odot$ to denote a general associative operator with neutral element $e_{\odot}$, defined either on $\mathbb{R}$ or $\mathbb{R}^n$. Given an array of variables in the program $\mathtt{x}$, we let $\bar{\mathtt{x}}$ denote the array of adjoint values in AD.

2.1 Non Parallel Procedure

[gray]0.31 **let** rs $=$ scan $\odot$ $e_\odot$ as

Let $\overline{\text{rs}}$ be the adjoint of rs, which is known at the current stage of automatic differentiation. We are then interested in calculating $\overline{\text{as}}$. From the formula

$$\begin{aligned} \text{rs}[0] &= \text{as}[0] \\ \text{rs}[i+1] &= \text{rs}[i] \odot \text{as}[i+1] \end{aligned}$$

We see that the value of rs$[i]$ depend on rs$[j]$ for $i > j$. For this reason we introduce another array of adjoint vectors $\overline{\text{x}}$, which is the same as $\overline{\text{rs}}$ but with the with contribution of the lower values to the higher values recorded. From the above formulas we then get

$$\begin{aligned} \overline{\text{x}}[n-1] &= \text{rs}[n-1] \\ \overline{\text{x}}[i-1] &= \frac{\partial(\text{rs}[i-1] \odot \text{as}[i])}{\partial \text{rs}[i-1]} \cdot \overline{\text{x}}[i] + \overline{\text{rs}}[i-1] \\ \overline{\text{as}}[0] &= \overline{\text{x}}[0] \\ \overline{\text{as}}[i] &= \frac{\partial(\text{rs}[i-1] \odot \text{as}[i])}{\partial \text{as}[i]} \cdot \overline{\text{x}}[i] \end{aligned}$$

From these formulas, we can write two for loops to calculate first $\overline{\text{x}}$ and then $\overline{\text{as}}$, however this would not preserve parallelism. We will now describe how to calculate these in a parallel way.

2.2 Parallelization

First of we note that, given the calculation of $\overline{\text{x}}$,

3 Implementation

4 Benchmark

To compare the efficiency of our implementation, we have run different benchmarks, to see how our implementation performs for different operations.

In our benchmarks, we test 4 different implementations.

1. Our own implementation.

2. An implementation of the same algorithm in Futhark written by Cosmin also for the [Bru+24] paper. This is to compare how efficient our implementation is.

3. The current procedure in Futhark, so we can compare the two different procedures.

4. The primal code, which performs the scan operation without any AD, to see how big the AD overhead is.

For the purpose of being able to compare with the Benchmarks of [Bru+24], we have done the same Benchmarks. We have also done further benchmarks

4.1 Comparison of PPAD implementation in Futhark and in compiler

Comparing the two PPAD implementations, we see that the compiler implementation runs slower or equal in 10/11 benchmarks, but that for 9 out of 11 benchmarks the runtime is at most double the runtime of the Futhark implementation.

The two exceptions to these are the matrix Multiplication for 5x5 matrices, and the vectorised addition, suggested the code can still be optimised for datatypes consisting of arrays. This is however contrasted by the vectorised multiplication which performs equally well in both implementations.

4.2 Comparison between PPAD and RMAD procedure

We see that PPAD-compiler implementation performs worse than the RMAD implementation, in all cases except for the linear operator on $\mathbb{R}^2$

$$(a, b) \odot (c, d) = (a + c + b \cdot d, b + d).$$

In a lot of cases this is expected as the vectorised operations, have extra optimization in [Bru+24], making them run faster. We still however see that slowdown is no more than a factor 2, for all benchmarks except 5x5 matrix multiplication, but that seems to an issue with the compiler implementation, rather than the procedure itself, as the futhark implementation is much close to the RMAD procedure.

5 Conclusion

References

[Bru+24] Lotte Maria Bruun et al. "Reverse-Mode AD of Multi-Reduce and Scan in Futhark". In: *Proceedings of the 35th Symposium on Implementation and Application of Functional Languages*. IFL '23. Braga, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400716317. DOI: 10.1145/3652561.3652575. URL: https://doi.org/10.1145/3652561.3652575.

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	7616	33828	170336	45008
50000000	37626	167955	848794	223939

Figure 1: Matrix Multiplication 5x5

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	960	6566	8263	6198
100000000	8986	64790	81734	61005

Figure 2: Matrix Multiplication 3x3

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	370	1552	2086	1579
100000000	3360	15019	20264	15101

Figure 3: Matrix Multiplication 2x2

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	188	614	750	746
100000000	1552	5637	6984	6698

Figure 4: Linear Function Composition

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	190	773	641	631
100000000	1560	7228	5919	5647

Figure 5: Linear Function Composition 2

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	369	2370	2744	1531
100000000	3248	23046	26746	14510

Figure 6: Function Composition

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	144	147	215	157
100000000	784	1270	1848	1325

Figure 7: Addition

	Primal	RMAD	PPAD-compiler	PPAD-futhark
1000000	1449	302	10265	1263
10000000	12995	2608	91315	11537

Figure 8: Vector Addition

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	146	147	215	565
100000000	1183	1270	1848	4866

Figure 9: Min

	Primal	RMAD	PPAD-compiler	PPAD-futhark
10000000	106	368	479	476
100000000	783	3229	4323	4012

Figure 10: Mul

	Primal	RMAD	PPAD-compiler	PPAD-futhark
1000000	1448	716	23582	24147
10000000	12968	6558	214282	214121

Figure 11: vecmul

[Pas+21] Adam Paszke et al. "Parallelism-preserving automatic differentiation for second-order array languages". In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*. FHPNC 2021. Virtual, Republic of Korea: Association for Computing Machinery, 2021, pp. 13–23. ISBN: 9781450386142. DOI: 10.1145/3471873.3472975. URL: https://doi.org/10.1145/3471873.3472975.