Master's Thesis Proposal

Simon van Hus
6147879
s.vanhus@students.uu.nl

December 10, 2022

1 Introduction

The aim of this research project is to built an implementation of reverse-mode automatic differentiation that – through program transformation – preserves parallelism structures present in the source program in the differentiated program. The goal is to implement this using a dual-numbers approach, and on top of Accelerate, a parallelized array language extension for Haskell [6].

Automatic differentiation (AD) is an approach for computers to differentiate programs as if they were mathematical functions. Differentiation is useful for application all throughout science, and being able to do it on an efficient basis therefor becomes very important. Current approaches can provide program transformations to create differentiated programs in polynomial time compared to the size of the input program. However, in these approaches using program transformation, complex structures in the program – like parallelism – are generally discarded. This is a shame, because many parallelized operations have parallelizable equivalents in the differentiated form. So discarding these structures means that the differentiated program can be much slower than the primal program, while it really doesn't have to be.

In the rest of this proposal, I'll go over the literature about this topic (in Section 2), the research questions and goals (in Section 3), and finally the planning going forward (in Section 4).

2 Literature

2.1 Forward and Reverse Mode

Automatic differentiation differentiates a program as a function $f : \mathbb{R}^n \to \mathbb{R}^m/$. In general, there's two major techniques for doing so [5, 2]. We have forward mode, which differentiates the program recursively using the chain rule. This means generating the derivatives in the same order of operations as the primal function, which is both intuitive and easy to implement.

Reverse mode, like the name suggests evaluates the program in reverse of the execution order. Rather than fixing the independent variables in the function, like in forward mode, reverse mode fixes the dependent variables and computes the so-called "adjoint" tangents which are achieved by performing a reverse pass over the computation tree.

The most important distinction between these two modes is what part of the Jacobian matrix they produce. As forward mode evaluates all dependent variables for a single dependent variable, effectively calculating a column in the Jacobian. This then also means that reverse mode – evaluating all dependent variables based on a single fixed independent variable – calculates a row of the Jacobian matrix of the function. This is important for efficient computation. Both forward-mode and reverse-mode can be calculated in polynomial time based on the size of the input program, however the dimensions of the Jacobian (or the number of independent and dependent variables) dictates which is more efficient. Namely, the derivative of a program or function $f : \mathbb{R}^n \to \mathbb{R}^m$ can be calculated more efficiently using forward mode AD when the function has (a lot) more outputs than it has inputs – so $m \gg n$, as only n sweeps will be necessary to calculate the full Jacobian. Conversely, reverse mode is more efficient when the number of inputs far outweigh the number of outputs, so $m \ll n$, requiring only m sweeps.

Reverse mode AD is especially current due to many applications in Artificial Intelligence using backpropagation, which can be considered a special case of reverse mode AD [1]. This is appropriate because many of these systems have many more inputs than outputs.

2.2 Dual-Numbers

One of the techniques to actually implement automatic differentiation is using dual-numbers. Dual-numbers is a technique where we replace scalar values by a tuple, like complex number. Where complex numbers exists of a real and imaginary part, dual numbers exist of a regular "primal" part, and a differentiated "dual" part [5].

It should be noted that in regular computation, these parts do not affect each other like how the real and imaginary parts of complex numbers might. Instead, the dual part of the number is updated for each part of the operations done on the primal part, to reflect the changes in either the forward or reverse derivative [4, 9]. Forward-mode on dual numbers is then fairly easy to understand, in the same way we calculate the primal expression, we can calculate the dual expression by using dual operations. Reverse-mode AD on dual numbers is a little more complicated, because we can't calculate the derivatives whilst we do the primal (forward) pass on the expression. Instead, we prepare the dual part of the numbers for a second backwards pass over the expression tree to calculate the derivative with. We find ourselves effectively performing a more generalized form of backpropagation [10] here, as we work our way back through the expression.

Especially for reverse-mode on dual-numbers, some additional steps need to be taken to make sure efficiency can be preserved. A notable example of this is preventing duplicate calculations. Duplicate calculations can happen quickly if we just naively work through an expression, calculating the parts we need when we need them. Instead, we it might be more efficient to calculate the dual expression in a "tape" or "Wengert list" [3]. The main downside of this is that if we express our expression like this, we'd lose any parallelism present in the program, as we're forced to work through the tape one item at the time.

2.3 Parallelism

Parallelism, especially on arrays, is also very current with the high interest in artificial intelligence. For both evaluating and training artificial neural networks, which is mostly matrix and vector multiplication, parallelism can provide a much appreciated performance increase. And while the backpropagation algorithm is a special case of reverse-mode differentiation, if we'd use the aforementioned dual-numbers approach with a tape, we'd lose the parallelism we had in the original program.

However, while not with tape, methods of maintaining parallelism in automatic differentiation are defintelitely possible. This can for example be achieved on second-order languages by paying special attention to the nesting of parallelism in the primal program, and applying special compile-time program transformations to get parallelized forward and reverse AD [7]. The main drawback of this being that we're limited to a second-order programming language.

Of course, we'd rather avoid restricting the language by allowing higher-order functions. This has also been done [8], using loop transformation, however doing this makes it hard to prove correctness, and doesn't use dual-numbers.

3 Research Questions and Goals

The main goal of my thesis will be to implement a dual-numbers program transformation for reverse-mode automatic differentiation, on top of the Accelerate array programming language for Haskell.

4 Planning

References

[1] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research 18*, 153 (2018), 1–43.

[2] Elliott, C. The simple essence of automatic differentiation. *Proc. ACM Program. Lang. 2*, ICFP (jul 2018).

[3] Griewank, A., and Walther, A. *Evaluating derivatives: principles and techniques of algorithmic differentiation.* SIAM, 2008.

[4] Krawiec, F., Jones, S. P., Krishnaswami, N., Ellis, T., Eisenberg, R. A., and Fitzgibbon, A. W. Provably correct, asymptotically efficient, higher-order reverse-mode automatic differentiation. *Proc. ACM Program. Lang. 6*, POPL (2022), 1–30.

[5] Margossian, C. C. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery 9*, 4 (2019), e1305.

[6] Marlow, S., et al. Haskell 2010 language report. *https://www.haskell.org/definition/haskell2010.pdf* (2010).

[7] Schenck, R., Rønning, O., Henriksen, T., and Oancea, C. E. Ad for an array language with nested parallelism. *arXiv preprint arXiv:2202.10297* (2022).

[8] Shaikhha, A., Fitzgibbon, A., Vytiniotis, D., and Peyton Jones, S. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang. 3*, ICFP (jul 2019).

[9] Smeding, T., and Vákár, M. Efficient dual-numbers reverse ad via well-known program transformations. *arXiv preprint arXiv:2207.03418* (2022).

[10] Wang, F., Zheng, D., Decker, J., Wu, X., Essertel, G. M., and Rompf, T. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages 3*, ICFP (2019), 1–31.