Master's Thesis Proposal

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

December 19, 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 therefore becomes very important. Current approaches can provide program transformations to create differentiated programs that have only constant overhead compared to the input program (for sequential programs). 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 does not have to be.

In the rest of this proposal, I will go over the research goals (in Section 2), the literature about this topic and relevant topics (in Section 3), the methodology I plan to apply for this project (in Section 4), and finally the planning going forward (in Section 5).

2 Research 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. This implementation will need to set itself apart by allowing for maintaining functional array parallelism from the primal program to the differentiated program.

This research goal is novel, because currently no such implementation exists for Accelerate, and the exact details of how such an implementation would work are yet unknown.

The research is mostly home in the domain of Computing Science, as automatic differentiation is a computing problem, however it is also relevant to Artificial Intelligence, because many AI implementations (like neural networks) rely on automatic differentiation to learn.

3 Literature

3.1 Automatic Differentiation

Automatic Differentiation is a way of turning a program $f : \mathbb{R}^n \to \mathbb{R}^m$ into its derivative f'. However, there are a couple of ways to differentiate f besides AD. One could hand-code the differentiated function if it is small enough, and if only that derivative is needed, this might be the fastest way. Another way is by using finite differences (also known as numeric differentiation), applying some small change ϵ to each input to find the effect on the output. Finite differences is easy to code, but is prone to floating point and truncation errors, and it can be slow for functions in high dimensions. Symbolic differentiation rewrites code to the differentiated version which gives exact results, however this is memory intensive, slow, and prone to expression swell. Finally, automatic differentiation creates a differentiated function f' without risk of expression swell or errors, and is highly applicable.

A major advantage of AD is that it can be implemented to handle higher-order functions and program flow, which is harder or impossible for the other differentiation methods. It can also do this efficiently, with a constant overhead on the differentiated program with regards to the running time of the original program.

In general, automatic differentiation (AD) comes in two flavours: forward mode and reverse mode [5, 2]. Both methods use the chain rule to differentiate a complex function by breaking it down into its smallest components and then recombing them to get the final result.

3.1.1 Forward Mode

In forward mode AD, or forward accumulation, we calculate f' by applying the chain rule repeatably to each subexpression. In a pen-and-paper differentiation this would look like Equation 1, where $\frac{\partial w_i}{\partial x}$ is the partial derivative of x on some step w_i, where step w_n would be equal to f itself.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial f}{\partial w_{n-1}} \left(\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x} \right) = \dots \tag{1}$$

Forward mode is pleasant to think about and implement because the calculation of the derivative follows the same 'path' as the regular (primal) expression. This is also expressed in the example in Table 1. A way to implement this is by using a technique called dual numbers, which I will discuss in Section 3.2.

The main drawback of forward mode is that for a function with n inputs and m outputs, we would need to run the AD over each input, so n times. This is especially problematic

Primal value	Derivative
$w_1 = x_1$	$\dot{w_1} = 1$ (seed)
$w_2 = x_2$	$\dot{w_2} = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w_3} = w_2 \cdot \dot{w_1} + w_1 \cdot \dot{w_2}$
$w_4 = \sin w_1$	$\dot{w_4} = \cos w_1 \cdot \dot{w_1}$
$w_5 = f = w_3 + w_4$	$\dot{w_5} = \dot{f} = \dot{w_3} + \dot{w_4}$

Table 1: Forward mode AD on a function $f(x_1, x_2) = \sin w_1 + w_1 w_2$, calculating $\frac{\partial f}{\partial x_1}$

Reverse Derivative
$\bar{w_5} = f = 1$ (seed)
$\bar{w_4} = \bar{w_5}$
$\bar{w_3} = \bar{w_5}$
$\bar{w_2} = \bar{w_2} \cdot w_1$
$\bar{w_1} = \bar{w_3} \cdot w_2 + \bar{w_4} \cdot \cos w_1$

Table 2: Reverse mode AD on a function $f(x_1, x_2) = \sin x_1 + x_1 x_2$, calculating $\frac{\partial f}{\partial \vec{x}}$

if $n \gg m$, which unfortunately comes up quite a lot, especially in machine learning.

3.1.2 Reverse Mode

Like the name suggests, reverse mode AD, or reverse accumulation, calculates the derivative f' of f in the reverse execution order. This hinges on the realization that while we can also fix the output variables, and instead calculate the gradient by calculating the differentiation in the opposite direction. So rather than $\frac{\partial f}{\partial x}$ for every x in the input, we calculate $\frac{\partial f}{\partial x}$ for every f in the output. When we can again express this in an equation, Equation 2, where w_i is step i in the primal calculation, where w_n would again be f. The reverse mode version of the example in Table 1 can be found in Table 2. It should also be noted that $\bar{w_i} = \frac{\partial f}{\partial w_i}$ is called "the adjoint" of w_i.

Now, with dual numbers AD, to actually get to calculating the reverse derivative, we first need to do a forward pass over the primal function to set up backpropagators. Then the reverse pass becomes just calling the final backpropagator with some seed value (usually 1). This means that if we have m outputs, we'll have m backpropagators to call, just like we needed to make n sweeps in forward mode for n inputs.

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial f}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \dots \tag{2}$$

Now, where forward mode required n passes to generate the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}^m$, reverse mode requires m passes. This means that if $n \gg m$ reverse mode works much more efficient than forward mode, and conversely if $n \ll m$ forward mode will be more efficient. Reverse mode can also be implemented using dual numbers, with a small adjustment (see Section 3.2).

Reverse mode AD is especially current, as it is a more generalized case of the backpropagation algorithm used in neural networks [1, 11]. The backpropagation algorithm is this specific case where, in general, $n \gg m$ for a function $f : \mathbb{R}^n \to \mathbb{R}^m$.

3.2 Dual Numbers

One of the techniques to actually implement automatic differentiation is using dual numbers. In dual numbers, we augment the "primal" scalars with a "dual" component. This dual component will store the partial derivative (or information relating to it). In a sense this dual component is similar to the imaginary component of a complex number, however it does not interact with the primal component like how the imaginary part interacts with the real part [5]. In general we find dual numbers are mostly represented as tuples, with the first element being the primal part, and the second element being the dual part.

For forward mode AD, the implementation with dual numbers is quite straightforward. In general, it can be easily implemented by operator overloading, where operators are rewritten to support dual numbers. Then we can simply replace the input variables to the function with these dual numbers, and as long as all operations in the function support dual numbers we will get the dual numbers representing the primal and derivative outcomes at the end.

Equation 3 provides a more concrete example of dual numbers for forward mode AD with an example multiplication, where $(x_i, \partial x_i)$ represents the dual number with the primal and dual part respectively.

$$(x_1, \partial x_1) \times (x_2, \partial x_2) = (x_1 \times x_2, x_1 \times \partial x_2 + x_2 \times \partial x_1) \tag{3}$$

Dual numbers for reverse mode AD are also possible, but they are slightly more complicated. The partial derivatives of steps in the expression are dependent on steps that happen further down, so we cannot immediately calculate the partial derivative. Instead, a solution would be to have the dual part be a function akin to a backpropagator [11, 10]. This means that the example of Equation 3 for forward mode, becomes the example in Equation 4 for reverse mode. Since ∂x_1 and ∂x_2 are backpropagator functions, they have the form $\partial x_i : \mathbb{R} \to c$, where c is the domain of the primal function ($\mathbb{R}^n$ in our examples), which is why they are presented with an argument d in Equation 4.

$$(x_1, \partial x_1) \times (x_2, \partial x_2) = (x_1 \times x_2, \lambda d.\, (\partial x_2\ (x_1 \times d) + \partial x_1\ (x_2 \times d))) \tag{4}$$

It should be clear now that reverse mode, with its advantage for calculating certain gradients, is also more complex to implement. For forward mode, the primal part and the dual part can be defined and calculated in one pass. This means that finding the forward mode derivative $\mathcal{F}(f)$ can be done in time complexity equal to f with some constant overhead, where f is the original primal function. Unfortunately, getting reverse mode AD on dual numbers to that level is a little more in depth. For starters, we effectively need to pass over the primal function twice: a forward pass to construct the backpropagators and a reverse pass to calculate the actual gradient. This complicates the matter greatly, because the structure of the function or program is much less clear on the reverse pass. Most importantly, as certain variables (or steps in the computation) are referenced multiple times, we will reach them that many times in the reverse pass, and risk calling their backpropagators multiple times as well. For large computations, this is unacceptable, as this problem compounds exponentially with the size of the program (each duplicate computation calls more duplicate computations). A way to solve this duplication issue is by storing the derivatives on a "tape", "trace", or Wengert list [3].

This tape can be constructed on the forward pass, and effectively topologically sorts the derivative of the expression [10, 4]. Now we can just calculate the backpropagators in the list one by one, to get the final derivative. The major downside of this is that we flatten out any parallelism present in the function, and turn it into a linear list.

3.2.1 Implementation Methods

There are two main ways to implement automatic differentiation in modern programming languages. First off, we have operator overloading, which extends the operators of a programming language to by adding support for use on new primitives or objects. Imperative languages like Python allow operator overloading by implementing operator methods like `__add__` to classes, which is then called by the `+` operator. In functional languages like Haskell, this can be done by extending instances like `Num` for use on new data types. Operator overloading is especially useful for forward mode AD, as it is generally easy to implement and intuitively provides tools for forward mode. For reverse mode, implementation using operator overloading is much harder, as we need to store every intermediate step and then still do the reverse pass.

Another major way to implement AD is by using source-code transformation. Like the name suggests, source-code transformation looks at the entirety of the source-code and produces some 'transformed' code, which in our case would be the differentiated program. This is generally done as a preprocessing step for the compiler, and allows for greater overview of the entire program to be differentiated. While source-code transformation can eliminate the need for a runtime tape to store intermediate values in, it is generally more complex to implement and complicates the compilation pipeline.

3.3 Functional Array Parallelism

To quickly process data, especially for example in neural networks, we use parallelism. Specifically for my research, we are interested in Functional Array Parallelism. Like the name suggests, this is parallelism on arrays in functional programming languages. This array parallelism is very important for implementations of neural networks, which generally consist of matrices and vectors that need to have functions applied to, or be multiplied together. The array parallelism allows us to exploit the architecture of hardware like GPUs to get much faster computation.

Functional array programming generally comes with two key functions. First off, a 'generate' function allows for creation of an array by providing the size of the array to create and either values or a function to populate the array with. As we're talking about functional array languages, we generally can't allow null-types in our arrays. Second, a 'fold' function allows an associative binary function to reduce one or more dimensions of the array to a single value. Getting the sum of an array is generally implemented using 'fold'. It should also be noted that many of these languages implement more specialized variants of 'generate' and 'fold', which can be more efficient than their generalized counterparts.

For my research I will be using the Accelerate parallelized array language, an extension on Haskell.

3.3.1 AD and Functional Array Parallelism

There are some examples of automatic differentiation implementations that support array parallelism.

First off, Amir Shaikhha et al. [9] show efficient automatic differentiation on a functional array programming language. They do this is for forward mode both with and without dual numbers, but prefer the use of direct source-to-source source-code transformations. A very interesting part of their paper is that they show how to calculate the gradient of a function using forward-mode AD in a way that's as efficient as reverse-mode. Furthermore, they also show some interesting work on loop fusion (and other transformations), which may prove useful, as fusing operations is a major part of Accelerate.

Secondly, Robert Schenck et al. [8] show a method of performing both forward and reverse mode automatic differentiation on a parallelized functional array language. Their array language only supports second-order functions, which allows them to eliminate the need for tape altogether. Their language also uses nested parallelism, which isn't supported by Accelerate. This paper goes very in-depth on the transformations of operations common in (parallelized) functional array languages, and how to implement them using dual numbers.

Finally, the Dex functional array programming language by Adam Paszke et al. [7] was designed with both automatic differentiation and parallel programming in mind. It performs reverse AD by using program transformations and reaches good performance doing so. It also supports higher-order functions. However they don't use dual numbers, which makes it hard to proof the correctness of their automatic differentiation.

4 Methodology

As briefly mentioned in Section 3, I'll be implementing a reverse mode automatic differentiation dual numbers approach in Accelerate for Haskell. The main idea here is to make use of Generalized Algebraic Data Types (GADTs) to represent program structure using dual numbers, which can then be used by source-code transformation to create differentiated programs. Using a standard "tape" will result in the computation losing it parallelization [10], so we need an alternative that preserves some of the program structure. While the exact details still need to be worked out, we discussed the use of a sort of hierarchical list to maintain program structure, or perhaps a directed acyclic graph with clear input and output nodes.

5 Planning

Phase two of this research project will start on Monday, December 19th 2023, and will finish on Sunday, July 30th 2023. After deciding on a method to maintain parallelism, the algorithm needs to be implemented and evaluated. Evaluation will be in terms of both speed and memory usage, in comparison to other reverse AD implementations. This first evaluation will undoubtedly be followed by changes, improvements, and optimizations to the algorithm. Finally, after the algorithm is finalized and evaluated again, the writing of the final paper happens.

The following schedule will provide a rough outline for how I am planning for the project to go:

December 2022 will focus of writing this proposal, and defining exactly how a differentiated program in Accelerate can maintain their parallelism. This month also contains a week of Christmas break (week 52).

January 2023 will focus on developing and implementing the first version of the AD algorithm.

February 2023 will focus on evaluation of, further development of, and improvement to the first version algorithm.

March 2023 will focus on optimization of the algorithm.

April 2023 will focus on evaluation and finalization of the final AD algorithm.

May 2023 will focus on starting the research paper.

June 2023 will focus on the research paper.

July 2023 will focus on finishing the research paper and the thesis defence presentation.

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] Paszke, A., Johnson, D., Duvenaud, D., Vytiniotis, D., Radul, A., Johnson, M., Ragan-Kelley, J., and Maclaurin, D. Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming. *arXiv preprint arXiv:2104.05372* (2021).

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

[9] 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).

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

[11] 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.