Efficient Differentiable Programming in a Functional Array-Processing Language

AMIR SHAIKHHA, University of Oxford, United Kingdom
ANDREW FITZGIBBON, Microsoft Research, United Kingdom
DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom
SIMON PEYTON JONES, Microsoft Research, United Kingdom

We present a system for the automatic differentiation (AD) of a higher-order functional array-processing language. The core functional language underlying this system simultaneously supports both source-to-source forward-mode AD and global optimisations such as loop transformations. In combination, gradient computation with forward-mode AD can be as efficient as reverse mode, and that the Jacobian matrices required for numerical algorithms such as Gauss-Newton and Levenberg-Marquardt can be efficiently computed.

CCS Concepts: • Mathematics of computing → Automatic differentiation; • Software and its engineering → Functional languages.

Additional Key Words and Phrases: Linear Algebra, Differentiable Programming, Optimising Compilers, Loop Fusion, Code Motion.

ACM Reference Format:

... in the summer of 1958 John McCarthy decided to investigate differentiation as an interesting symbolic computation problem, which was difficult to express in the primitive programming languages of the day. This investigation led him to see the importance of functional arguments and recursive functions in the field of symbolic computation. From Norvig [Norvig 1992, p248].

1 INTRODUCTION

Forward-mode Automatic Differentiation is relatively straightforward, both as a runtime technique using dual numbers, or as a source-to-source program transformation. However, forward-mode AD is usually considered wildly inefficient as a way to compute the gradient of a function, because it involves calling the forward-mode AD function \( n \) times — and \( n \) may be very large (e.g. \( n = 10^6 \)).

This has led to a tremendous amount of work on reverse-mode AD. As a source-to-source transformation, reverse-mode AD is characterised by the necessity to maintain temporary variables holding partial results, to be consumed during a “reverse pass” of gradient computation. Modern

Authors’ addresses: Amir Shaikhha, University of Oxford, United Kingdom, amir.shaikhha@cs.ox.ac.uk; Andrew Fitzgibbon, Microsoft Research, United Kingdom, awf@microsoft.com; Dimitrios Vytiniotis, DeepMind, United Kingdom, dvytin@google.com; Simon Peyton Jones, Microsoft Research, United Kingdom, simonpj@microsoft.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2019 Copyright held by the owner/author(s).
2475-1421/2019/8-ART97
https://doi.org/10.1145/3341701
systems devote considerable effort (e.g., checkpointing) to optimizing the recompute/store tradeoff of these temporaries.

Our key contribution is this: we start from the “wildly inefficient” loop in which the forward-mode function is called \( n \) times, and demonstrate that it can be made efficient simply by applying a collection of non-AD-specific compile-time optimising transformations. In fact, our optimisations are sufficiently aggressive to generate code that computes the gradient with efficiency that is sometimes better than standard reverse-mode techniques. Moreover, the entire pipeline is much simpler: there is no need to grapple with the complexities of reverse mode — it simply falls out from the result of optimisation.

More specifically, our contributions are as follows:

• We introduce a small functional array language (Section 3), and describe forward-mode automatic differentiation as a simple source-to-source program transformation (Section 4).

• We introduce a collection of optimising transformations, none of them AD-specific, for our language (Section 5).

• For some small but realistic benchmarks, we show that our transformations suffice to generate extremely efficient code, starting from a naïve loop that repeatedly calls the forward derivative of the function. We compare our results with those of other AD systems (Section 6).

We discuss related work in Section 7.

2 BACKGROUND

Automatic Differentiation (AD) systematically applies the chain rule, and evaluates the derivatives for the primitive arithmetic operations (such as addition, multiplication, etc.). One of the key properties of AD is the constant-time overhead of the differentiated program with respect to the original code; not being careful about sharing during the process of differentiation, can lead to code explosion [Baydin et al. 2015b].

There are two main modes for implementing AD. Forward-mode computes the derivative part (tangent part) alongside the original computation while making a forward pass over the program. Reverse-mode makes a forward pass to compute the original part of the program, followed by a backward pass for computing the derivative part (adjoint part). Consider a program in ANF [Flanagan et al. 1993]:

\[
f(x_1, \ldots, x_n) = \\
\text{let } v_1 = e_1 \\
\vdots \\
\text{let } v_n = e_n \\
v_n
\]

To compute the derivative of this function using the forward-mode AD, we associate a tangent variable to each variable \( v_i \), denoted by \( \tilde{v_i} \). Each tangent variable is computed as \( \tilde{v_i} = \tilde{x_1} \times \frac{\partial v_i}{\partial x_1} + \ldots + \tilde{x_n} \times \frac{\partial v_i}{\partial x_n} \). In order to compute the partial derivative of \( f \) with respect to \( x_i \), we assign \( \tilde{x_i} = 1 \) and \( \tilde{x_j} = 0 \) for \( i \neq j \), and call the transformed code.

Reverse-mode AD computes the \( n \) partial derivatives simultaneously, rather than in two different iterations. To compute the derivative of this function using this approach, we associate an adjoint variable to each variable \( v_i \), denoted by \( \tilde{\tilde{v}} \), which is computed as \( \tilde{\tilde{v}} = \frac{\partial y}{\partial v_i} \). As a result, if we are interested in computing the partial derivative of function \( f \) with respect to \( x_1 \), we have to compute the value of \( \tilde{\tilde{x}} \). Similarly, if we are interested in the partial derivative of this function with respect
to $x_2$, we have to compute the value of $\tilde{x}_2$. To do so, we have to apply the chain rule in the reverse order.

Generally speaking, forward and reverse-mode compute a column and a row, respectively, of the full Jacobian matrix $J$ at each invocation. More precisely, for a function with an input vector of size $n$ and an output vector of size $m$, the forward mode approach computes a column vector of size $m$, and the reverse mode computes a row vector of size $n$ (see Figure 1).

For a class of optimisation problems, such as various computer vision problems using the Levenberg-Marquardt algorithm [Levenberg 1944; Marquardt 1963; Moré 1978], one is required to compute the full Jacobian matrix. In such cases, neither of the two techniques perform efficiently. To compute the full Jacobian matrix, both forward and reverse-mode techniques must iterate over either the columns or the rows of the Jacobian matrix, respectively. Given that both approaches have a constant overhead over the original computation, the forward mode technique is more efficient for computing the full Jacobian matrix when $m \gg n$, whereas the reverse mode AD is more efficient when $n \gg m$. However, when $n$ and $m$ are in the same range, it is not clear which approach performs better. Moreover:

- By carefully examining the body of the loops needed for computing the full Jacobian matrix, one can observe that many computations are loop-invariant and are unnecessarily performed multiple times. Thus, there is a lost opportunity for loop-invariant code motion for hoisting such expressions outside the loop, thus asymptotically improving the performance (cf. the NNMF and Bundle Adjustment experiments in Section 6).
- Furthermore, while the result of automatic differentiation is known to have only a constant factor more arithmetic operations than the original program, the constant can be significant; this overhead can have a dramatic impact on the run-time performance in practice. More specifically, in applications involving the manipulation of vectors, many intermediate vectors are allocated that can be removed. The optimisation for eliminating such intermediate vectors is known as deforestation [Coutts et al. 2007; Gill et al. 1993; Svenningsson 2002; Wadler 1988] or loop fusion in the functional programming community. This optimisation opens the door for many other optimisations such as normalising loops that are iterating over sparse vectors with a single non-zero element into a single statement (cf. Example 5 in Section 5).

3 OVERVIEW
In this section, we start with an overview of the compilation process in $d\bar{f}$, which is shown in Figure 2. This figure demonstrates the position of $d\bar{f}$ with respect to existing AD tools.
Fig. 2. Compilation process in different AD systems. The solid arrows correspond to the pipeline used in \( \tilde{d}\f \).

d\( \tilde{f} \) starts from a program written in a high-level linear algebra library, called \( \tilde{M} \) (Section 3.2). This program is lowered into its implementation in a higher-order functional language with array support, called \( \tilde{F} \) (Section 3.1). If a part of the program requires computing differentiation (which are specified by using high-level differentiation API exposed by \( \tilde{d}\f \), as mentioned in Section 4.1) \( \tilde{d}\f \) uses AD transformation rules (Section 4.2) for transforming the involved expressions into their differentiated form.

Finally, after applying several simplifications such as loop fusion, partial evaluation, data layout transformation, etc. (Section 5) the differentiated program is transformed into low-level C code. The generated C code uses efficient stack-discipline memory management by using the destination-passing style (DPS) technique [Shaikhha et al. 2017]. Alternatively, one can use other array programming languages such as Futhark [Henriksen et al. 2017] and SAC [Grelck and Scholz 2006] as the target language for differentiated \( \tilde{F} \) programs.

Next, we present the core functional language used in \( \tilde{d}\f \), on top of which we define source-to-source AD transformation and simplification rules.

3.1 \( \tilde{F} \)

\( \tilde{F} \) (pronounced as F smooth) is a subset of \( \tilde{F}\# \), an ML-like higher-order functional programming language. It is designed to be expressive enough to make it easy to write array-processing workloads, while simultaneously being restricted enough (e.g., avoiding partially applied functions and returning functions from lambdas which are enforced by the (T-App) and (T-Abs) typing rules, respectively) in order to compile it to code that is as efficient as hand-written C, with very simple and efficient memory management [Shaikhha et al. 2017].

Figure 3 shows the abstract syntax (parentheses can be used as necessary), type system, and several built-in functions of \( \tilde{F} \). \( \bar{x} \) and \( \bar{e} \) denote one or more variables and expressions, respectively, which are separated by spaces, whereas, \( \bar{T} \) represents one or more types which are separated by ⇒. In addition to the usual \( \lambda \)-calculus constructs (abstraction, application, and variable access), \( \tilde{F} \) supports let binding and conditionals.
Efficient Differentiable Programming in a Functional Array-Processing Language

\[
e ::= e \mid \text{fun } x \rightarrow e | x\quad - \text{Application, Abstraction, and Variable Access}
\]
\[
| n \mid i \mid N\quad - \text{Scalar, Index, and Cardinality Value}
\]
\[
| c\quad - \text{Constants (see below)}
\]
\[
| \text{let } x = e \text{ in } e\quad - \text{(Non-Recursive) Let Binding}
\]
\[
| \text{if } e \text{ then } e \text{ else } e\quad - \text{Conditional}
\]
\[
T ::= M\quad - \text{(Non-Functional) Expression Type}
\]
\[
| T \Rightarrow M\quad - \text{Function Types (No Currying)}
\]
\[
M ::= \text{Num}\quad - \text{Numeric Type}
\]
\[
| \text{Array}_<M>\quad - \text{Vector, Matrix, ... Type}
\]
\[
| M \times M\quad - \text{Pair Type}
\]
\[
| \text{Bool}\quad - \text{Boolean Type}
\]
\[
\text{Num ::= Double} \mid \text{Index} \mid \text{Card}\quad - \text{Scalar, Index, and Cardinality Type}
\]

Typing Rules:

\[
\frac{e_0 : T \Rightarrow M \quad \bar{e} : \bar{T}}{(T-App)\quad e_0 \bar{e} : M}
\]
\[
\frac{\Gamma \cup \bar{x} : \bar{T} \vdash e : M}{(T-Abs)\quad \Gamma \vdash \lambda \bar{x} . e : T \Rightarrow M}
\]
\[
\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2}{(T-Let)\quad \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}
\]
\[
\frac{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : M}{(T-If)\quad e_1 : \text{Bool} \quad e_2 : M \quad e_3 : M}
\]

Scalar Function Constants:

\[
+ \mid - \mid \times \mid / \mid **\quad : \text{Num, Num} \Rightarrow \text{Num}
\]
\[
\sin \mid \cos \mid \tan \mid \log \mid \exp\quad : \text{Num} \Rightarrow \text{Num}
\]
\[
| > | < | == | <>\quad : \text{Num} \Rightarrow \text{Num} \Rightarrow \text{Bool}
\]
\[
&\& | ||\quad : \text{Bool} \Rightarrow \text{Bool} \Rightarrow \text{Bool}
\]
\[
!\quad : \text{Bool} \Rightarrow \text{Bool}
\]

Vector Function Constants:

\[
\text{build}: \text{Card} \Rightarrow (\text{Index} \Rightarrow M) \Rightarrow \text{Array}_<M>
\]
\[
\text{get}: \text{Array}_<M> \Rightarrow \text{Index} \Rightarrow M
\]
\[
\text{ifold}: (M \Rightarrow \text{Index} \Rightarrow M) \Rightarrow M \Rightarrow \text{Card} \Rightarrow M
\]
\[
\text{length}: \text{Array}_<M> \Rightarrow \text{Card}
\]

Pair Function Constants:

\[
\text{pair}: M_1 \Rightarrow M_2 \Rightarrow M_1 \times M_2
\]
\[
\text{fst}: M_1 \times M_2 \Rightarrow M_1
\]
\[
\text{snd}: M_1 \times M_2 \Rightarrow M_2
\]

Syntactic Sugar:

\[
\begin{align*}
e_0[e_1] &= \text{get } e_0 e_1 \quad \text{Matrix} &= \text{Array}_<\text{Array}_<\text{Double}>>
\end{align*}
\]
\[
\begin{align*}
(e_0, e_1) &= \text{pair } e_0 e_1 \quad \text{DoubleD} &= \text{Double} \times \text{Double}
\end{align*}
\]
\[
\begin{align*}
e_1 \text{ bop } e_2 &= \text{bop } e_1 e_2 \quad \text{VectorD} &= \text{Array}_<\text{Double} \times \text{Double}>
\end{align*}
\]
\[
\begin{align*}
\text{Vector} &= \text{Array}_<\text{Double}>
\end{align*}
\]
\[
\begin{align*}
\text{MatrixD} &= \text{Array}_<\text{Array}_<\text{Double} \times \text{Double}>>
\end{align*}
\]

Fig. 3. The syntax, type system, and function constants of the core $\tilde{F}$.

$\tilde{F}$ supports array programming by defining the following built-in functions: build for producing arrays; ifold for iteration for a particular number of times (from 0 to $n-1$) while maintaining a state across iterations; length to get the size of an array; and get to index an array.

One of the key features of $\tilde{F}$ is its support for both source-to-source automatic differentiation and global optimisations such as loop-invariant code motion and loop fusion. The transformations required for automatic differentiation are presented in Section 4.2, and the ones for optimisation and simplification are shown in Section 5.

Next, we show how a Linear Algebra library can be defined on top of $\tilde{F}$.
Table 1. Equivalent operations in Matlab, R, NumPy, and $\tilde{M}$.

<table>
<thead>
<tr>
<th>Matlab</th>
<th>R</th>
<th>NumPy</th>
<th>$\tilde{M}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A * B</td>
<td>A %*% B</td>
<td>A.dot(B)</td>
<td>matrixMult A B</td>
</tr>
<tr>
<td>A + B</td>
<td>A + B</td>
<td>A + B</td>
<td>matrixAdd A B</td>
</tr>
<tr>
<td>A'</td>
<td>t(A)</td>
<td>A.T</td>
<td>matrixTranspose A</td>
</tr>
<tr>
<td>ones(n, m)</td>
<td>matrix(1, n, m)</td>
<td>ones((n, m))</td>
<td>matrixOnes n m</td>
</tr>
<tr>
<td>zeros(n, m)</td>
<td>matrix(0, n, m)</td>
<td>zeros((n, m))</td>
<td>matrixZeros n m</td>
</tr>
<tr>
<td>eye(n)</td>
<td>diag(n)</td>
<td>eye(n)</td>
<td>matrixEye n</td>
</tr>
</tbody>
</table>

3.2 $\tilde{M}$

$\tilde{M}$ is a functional Linear Algebra library, mainly inspired by MATLAB and R, programming languages which are heavily used by data analysts. By providing high-level vector and matrix operations, $\tilde{M}$ frees the users from low-level details and enables them to focus on the algorithmic aspects of the problem in hand.

$\tilde{M}$ is simply a $\tilde{F}$ library, but it can also be thought of as an embedded domain-specific language (EDSL) [Hudak 1996]. Figure 4 demonstrates a subset of $\tilde{M}$ operations which are defined as functions in $\tilde{F}$. This library is expressive enough for constructing vectors and matrices, element-wise operations, accessing a slice of elements, reduction-based operations (computing the sum of vector elements), matrix transpose, and matrix multiplication.\(^{1}\) Supporting more sophisticated operations such as matrix determinant and matrix decomposition is beyond the scope of the current paper, and we leave it for the future. As discussed before, $\tilde{M}$ is inspired by MATLAB and R. As a result, there is a mapping among the constructs of $\tilde{M}$ and these matrix-based languages. Hence, it is easily possible to translate a program written in one of these languages to $\tilde{M}$. Table 1 demonstrates the mapping among a subset of the constructs of MATLAB, R, NumPy and $\tilde{M}$.

**Example 1.** Assume that we have a matrix $M$ and two vectors $u$ and $v$ (which are represented as column matrices and are independent of $M$). Based on matrix calculus one can prove that $\frac{\partial}{\partial M}(uMv^T) = u^Tv$. However, computing the differentiated version of this function using forward-mode AD tools requires multiple iterations over the differentiated program for every element in the matrix $M$. By using the reverse-mode AD, one can invoke the differentiated function only once, and the adjoint parts of the input matrix $M$ will be filled in. We will show in Section 5 that $d\tilde{f}$ derives the gradient of this expression with respect to $M$, resulting in an expression equivalent to $u^Tv$. This optimises away multiple iterations over the differentiated program for each element of matrix $M$, in contrast to the existing AD tools based on the forward-mode AD technique.

For the moment, we only show how the matrix expression $uMv^T$ is expressed in $\tilde{M}$:

```haskell
let f = fun u M v -> let um = vectorToMatrix u let vt = matrixTranspose (vectorToMatrix v) let m = matrixMult um (matrixMult M vt) m[0][0]
```

The last expression is for accessing the single scalar element of a $1 \times 1$ matrix.

\(^{1}\)We have duplicated identical implementations for functions such as vectorMap and matrixMap (and similarly for vectorMap2 and matrixMap2) because of the restrictions imposed by $\tilde{F}$ [Shaikhha et al. 2017]. More specifically, the C code generation process does not handle polymorphic functions. Hence, we need to specify two different monomorphic functions with the same implementation for such functions.
Fig. 4. A subset of $\tilde{\mathcal{M}}$ constructs defined in $\tilde{\mathcal{F}}$.

4 DIFFERENTIATION

In this section, we show the differentiation process in $\tilde{\mathcal{D}}$. First, we start by the high-level API exposed by $\tilde{\mathcal{D}}$ to the end users. Then, we show how $\tilde{\mathcal{D}}$ uses automatic differentiation behind the scenes for computing derivatives. Finally, we present the optimisations offered by $\tilde{\mathcal{D}}$, and we demonstrate how $\tilde{\mathcal{D}}$ can use these optimisations to deduce several matrix calculus identities.

4.1 High-Level API

For computing the derivative of an arbitrary function, $\tilde{\mathcal{D}}$ provides the deriv construct. This construct can be better thought of as a macro, which is expanded during compilation time. The expanded expression includes the expression of the original computation, which is given as the first argument (and can be an arbitrary scalar, vector, or matrix expression), and the derivative
of this expression with respect to the variable given as the second argument, referred to as the independent variable. Note that one can easily compute the derivative of an expression with respect to a list of free variables by multiple invocation of the deriv construct.

Figure 5 shows the implementation of the deriv construct (denoted by the \( G[e]_x \) compile-time transformation, corresponding to computing the derivative of e with respect to x). First, deriv constructs a lambda function which has the free variables of the given expression as its input parameters.\(^2\) The produced lambda function is given as input to source-to-source automatic differentiation (denoted by \( D[] \)), which can handle expressions of arbitrary type as explained later in Section 4.2. The differentiated function is applied to the dual number encoding of all the free variables (using the \( A_i[] \) compile-time transformation, corresponding to the dual number encoding of a tensor expression of rank \( i \)). Based on the type of the independent variable, the result of this applied function will be the result scalar value, or an element of the result vector or matrix.

If the free variable is different than the input variable with respect to which we are differentiating (i.e., the independent variable), the derivative part is a zero scalar, vector, or matrix. Otherwise, the derivative part is a one-hot encoding scalar, vector, or matrix. If the independent variable has a scalar type, deriv returns the applied function. However, if the independent variable has a vector type, deriv constructs a vector with the same number of elements as the independent variable. For computing the \( r \)th element of the result vector, the corresponding input vector is a one-hot encoding with a single one at the \( r \)th position. The situation is similar for an independent variable with a matrix type; the corresponding one-hot encoding matrix has a single one at the \( r \)th row and \( c \)th column.

**Example 2.** Let us assume that we would like to compute the derivative of a program computing the cosine function with respect to its input:

\[
\cos a
\]

The derivative of this program at point \( a \) is represented as follows:

\[
\text{snd} \left( \text{deriv} \left( \cos a \right) a \right)
\]

This expression is transformed into the following expression after expanding the deriv macro:

\[
\text{snd} \left( \left( D[\text{fun} a \rightarrow \cos a] \right) (a, 1) \right)
\]\\

Furthermore, \( d\mathcal{F} \) provides three additional differentiation constructs, inspired by AD tools such as DiffSharp [Baydin et al. 2015a]: 1) \textbf{diff} computes the derivative of a function, from a real number to a real number, with respect to its input, 2) \textbf{grad} computes the gradient of a function, from a vector of real numbers to a real number, with respect to its input vector, and 3) \textbf{jacob} computes the Jacobian matrix of a vector-valued function, a function from a vector of real numbers to a vector of real numbers, with respect to its input vector. Figure 6 demonstrates how these high-level differentiation constructs are defined in terms of the source-to-source AD transformation construct \( D \).

**Example 2 (Continued).** For the previous example, if we would like to use the \textbf{diff} construct, first we have to define the following function:

\[
g = \text{fun} x \rightarrow \cos(x)
\]

The derivative of this function at point \( a \) is represented as follows:

\[\text{deriv} \text{ only handles expressions of scalar, vector, or matrix type, and cannot be used for function types. The same restriction applies for the extracted free variables, but there is no restriction on the type of other sub-expressions.} \]
\[ \text{deriv } x = G[e]_x \]

\[
G[e]_x =
\begin{align*}
\text{if } x & \text{ is } \text{Double} & (\mathcal{D}[\text{fun } v_i \rightarrow e]) A_0[v_i]_x \\
\text{if } x & \text{ is } \text{Vector} & \text{build}(\text{length } x) (\text{fun } r \rightarrow (\mathcal{D}[\text{fun } v_i \rightarrow e]) A_1[v_i]_{x,r}) \\
\text{if } x & \text{ is } \text{Matrix} & \text{build}(\text{matrixRows } x) (\text{fun } r \rightarrow \\
& & \text{build}(\text{matrixCols } x) (\text{fun } c \rightarrow (\mathcal{D}[\text{fun } v_i \rightarrow e]) A_2[v_i]_{x,r},c))
\end{align*}
\]

\textbf{where}

\[
\mathcal{V}_i = fvs(e)
\]

\[
A_0[x]_x = (x, 1) \\
A_0[v]_x = (v, 0)
\]

\[
A_1[x]_{x,r} = \text{vectorZip } x (\text{vectorHot(length } x) r) \\
A_1[v]_{x,r} = \text{vectorZip } v (\text{vectorZeros(length } v))
\]

\[
A_2[x]_{x,r,c} = \text{matrixZip } x (\text{matrixHot(matrixRows } x) (\text{matrixCols } x) r c) \\
A_2[v]_{x,r,c} = \text{matrixZip } v (\text{matrixZeros(matrixRows } v) (\text{matrixCols } v))
\]

Fig. 5. Implementation of the \text{deriv} construct as a source-to-source transformation pass.

<table>
<thead>
<tr>
<th>Oper.</th>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{diff}</td>
<td>(\text{Double} \Rightarrow \text{Double}) \Rightarrow \text{Double} &amp; \text{fun } f x \rightarrow \mathcal{D}[f] (x, 1)</td>
<td></td>
</tr>
</tbody>
</table>
| \text{grad} | (\text{Vector} \Rightarrow \text{Double}) & \Rightarrow \text{Vector} \Rightarrow \text{Vector} & \text{fun } f v \rightarrow \\
& & \text{build}(\text{length } v) (\text{fun } i \rightarrow \\
& & \mathcal{D}[f] (\text{vectorZip } v (\text{vectorHot(} \text{length } v) i)) \\
| \text{jacob} | (\text{Vector} \Rightarrow \text{Vector}) & \Rightarrow \text{Vector} \Rightarrow \text{Matrix} & \\
& & \\

\[ \text{snd } ((\text{diff } g) \ a) \]

which is expanded to the following program:

\[ \text{snd } (\mathcal{D}[g] (a, 1)) \]

\[ \text{Table 2. Different types of matrix derivatives.} \]

\[
\begin{array}{c|c|c|c}
\text{Input Type} & \text{Output Type} & \text{Scalar} & \text{Vector} & \text{Matrix} \\
\hline
\text{Scalar} & \text{diff} & \text{vdiff} & \text{mdiff} \\
\text{Vector} & \text{grad} & \text{jacob} & - \\
\text{Matrix} & \text{mgrad} & - & - \\
\end{array}
\]

\[ \text{Table 2 summarizes different matrix derivatives, and how they can be computed using our high-level API. Note that the definition of vdiff and mdiff is similar to diff, and the definition of mgrad is similar to grad and jacob (cf. Figure 6). Note that the deriv construct subsumes all these operators.} \]

\[ \Delta \]

One key advantage of defining different matrix derivatives in terms of automatic differentiation is that one no longer needs to define the matrix calculus derivative rules for all different combinations shown in Table 2. Instead these rules can be deduced automatically from the automatic differentiation rules defined for scalar values. Moreover, even the algebraic identities for matrix derivative can be deduced by using the simplification rules presented in Section 5.

Next, we present the source code transformation required for applying automatic differentiation rules.

4.2 Source-to-Source Automatic Differentiation

d\tilde{f} relies on source-to-source translation for implementing forward-mode automatic differentiation. Each expression is converted into an expression containing both the original computation, together with the derivative computation, a.k.a. the dual number technique. The scalar expressions are transformed into a pair of values, the original computation and the derivative computation. The vector expressions are transformed into vectors containing tuple expressions, instead of scalar expressions. The situation is similar for higher-rank tensors such as matrices.

The rules for automatic differentiation are demonstrated in Figure 7. \(D[e]\) specifies the AD translation for expression \(e\). A variable \(y\) is translated as \(\mathbf{\hat{y}}\), emphasizing that the translated variable keeps the derivative part as well (D-Abs, D-Var, and D-Let). \(P[e]\) is a shorthand for extracting the original computation from the translated term \(D[e]\), while \(E[e]\) is a shorthand for accessing the derivative part. Note that in order to avoid redundant computation and code explosion, especially for the differentiation rules such as the product rule (D-Mult), the arguments are bound to a new variable.

Constructing an array is differentiated as an array with the same size, however, the way that each element of the array is constructed is differentiated (D-Build). Differentiating an iteration results in an iteration with the same number of iterations, and with the initial state and the next state function both differentiated (D-IFold). The differentiation of the length and indexing an array, is the same as the length and indexing the differentiated array, respectively (D-Length and D-Get).

Differentiating a pair of elements results in the pair of differentiated elements (D-Pair). Similarly, differentiating the projection of a pair, is the projection of the differentiated pair (D-Fst, D-Snd). For other scalar-valued functions, the differentiation rules are similar to the corresponding rules in mathematics.

Example 2 (Continued). In the previous example, based on the automatic differentiation rules, the differentiated program would be as follows:

\[
\mathbf{\hat{g}} = \text{fun } \mathbf{\hat{x}} \rightarrow -\text{snd} (\mathbf{\hat{x}}) \ast \sin (\text{fst} (\mathbf{\hat{x}}))
\]

Based on the definition of the \texttt{diff} construct, we have to use the AD version of the function (i.e., \(g\)) and assign 1 to the derivative part of the input. So the value of \(\cos'\) for the input \(a\) is computed as follows:

\[
\begin{align*}
\text{snd} ((\text{diff } g) a) \quad &\Rightarrow \quad \text{snd} (D[[g]] (a, 1)) \quad \Rightarrow \quad \text{snd} (\hat{g} (a, 1)) \quad \Rightarrow \\
&\Rightarrow \quad -\text{snd} ((a, 1)) \ast \sin (\text{fst} ((a, 1))) \quad \Rightarrow \quad -1 \ast \sin (a) \quad \Rightarrow \quad -\sin (a)
\end{align*}
\]

Similarly, we can compute the partial derivatives of a given function, by setting the desired derivative part to one, and the rest of derivatives to zero. This process is illustrated in the next example.

Example 3. Assume that we would like to compute the partial derivative of the expression \(a \ast b\) with respect to \(a\), which is represented as follows in F:

\[
\text{snd} (\text{deriv} (a \ast b) a)
\]
This expression is expanded as follows:

\[
\text{snd} \left( \mathcal{D}\left[ \text{fun} \ a \ b \rightarrow a \ast b \right] (a, 1) (b, 0) \right)
\]

Note that the derivative part of the second input is set to 0. Similar to the previous example, the result is as follows:

\[
\text{snd} \left( \left( \text{fun} \ a \ b \rightarrow \left( \text{fst} \ (a) \ast \text{fst} \ (b), \ \text{fst} \ (a) \ast \text{snd} \ (b) + \ \text{snd} \ (a) \ast \text{fst} \ (b) \right) \right) (a, 1) (b, 0) \right)
\]

which is evaluated as follows:

\[
\text{snd} \left( (a \ast b, 1 \ast b + a \ast 0) \right) \leadsto 1 \ast b + a \ast 0 \leadsto b
\]

It is important to note that d\(\tilde{f}\) performs many of the evaluation steps shown for the previous examples during compilation time, i.e., performs partial evaluation. Section 5 gives more details on the optimisations and simplifications offered by d\(\tilde{f}\).

### 4.3 Perturbation Confusion and Nested Differentiation

In several problems such as computing the Hessian matrix, one requires to compute the differentiation of a differentiated program. In such cases, one should be careful dealing with tangent parts. We demonstrate this problem in the next example.

**Example 4.** Here is the classical example showing the perturbation confusion problem:

\[
\frac{\partial}{\partial x} \left( x \frac{\partial x + y}{\partial y} \right)
\]

This expression should be evaluated to 1 at every point. However, an AD tool can mistakenly evaluate this expression to 2. This is because of confusing the tangent part (perturbation) of the free variable \(x\), while computing the inner derivative. This is known as the *perturbation confusion* problem in the AD literature.

If one uses the differentiation API of Figure 6, the perturbation confusion problem appears. In order to avoid this problem, the `deriv` macro needs to be used. The macro expansion of the `deriv` operator can be thought of as a preprocessing step that binds each of the perturbations to a different variable [Siskind and Pearlmutter 2005]. Then, it is the responsibility of the \(\tilde{F}\) programming language implementation (e.g., using alpha renaming as mentioned in [Siskind and Pearlmutter 2008]) to avoid the perturbation confusion problem. We demonstrate this fact on our running example.

**Example 4 (Continued).** The previous expression is implemented as follows in the \(\tilde{F}\) language:

```latex
fun \ x \ y \rightarrow
    snd (deriv (x ^ (snd (deriv (x + y) y)))) x
```

After expanding the inner `deriv` macro, the following expression is derived:
fun x y ->
snd (
let t1 = snd (
  (fun x y -> (fst x) + fst y, snd x + snd y)) (x, 0) (y, 1)
) x * t1)
Note that the variable \( t_1 \) is created in order to avoid code explosion that can result from the product rule (cf. Section 4.2). Expanding the outer \( \text{deriv} \) macro results in the following expression:

```adam
fun x y ->
  snd (fun \( \vec{x} \) \( \vec{y} \) ->
    let \( \vec{t}_1 \) = snd (fun \( \vec{x} \) \( \vec{y} \) ->
      ((fst (fst (\( \vec{x} \) ))) + fst (fst (\( \vec{y} \) ))) ,
      (fst (snd (\( \vec{x} \) ))) + fst (snd (\( \vec{y} \) ));
     snd (snd (\( \vec{x} \) )) + snd (snd (\( \vec{y} \) ))
    ) (\( \vec{x} \) , (0, 0)) (\( \vec{y} \) , (1, 0))
  )
  (fst (\( \vec{x} \) ) * fst (\( \vec{t}_1 \) ),
  snd (\( \vec{x} \) ) * fst (\( \vec{t}_1 \) ) +
  fst (\( \vec{x} \) ) * snd (\( \vec{t}_1 \))
)(x, 1) (y, 0)
)
```

Note that this macro expansion results in the inner perturbation variables \( \vec{\vec{x}} \) and \( \vec{\vec{y}} \) which are different from the outer variables \( \vec{x} \) and \( \vec{y} \). This different naming results in avoiding the perturbation confusion problem. Finally, partially evaluating the inner expression results in the following expression:

```adam
fun x y ->
  1
```

5 EFFICIENT DIFFERENTIATION

In this section, we show how \( dF \) achieves efficient differentiable programming. First, we show several transformation rules applicable on \( \tilde{F} \) expressions. We show how these transformation rules are used to derive matrix-algebraic identities, in the level of \( \tilde{F} \) expressions. Then, we show how we generate C code from \( \tilde{F} \) expressions for more efficient memory management.

5.1 Transformation Rules

There are various algebraic identities that one can define for \( \tilde{F} \). Based on these identities, vector and matrix-based programs, as well as differentiated programs can be heavily optimised. Figure 8 shows a set of optimisations defined for \( \tilde{F} \). Through examples, we show how these rewrite rules can discover vector and matrix-level algebraic equalities.

There are various optimisations defined for scalar operations based on the ring structure of addition and multiplication, which are shown in Figure 8b. Note that other ring-based algebraic identities, such as associativity and commutativity, do not appear directly in the list of rules that \( dF \) applies. This is because they do not necessarily improve the performance, unless they are combined with other rewrite rules.

As \( F \) is based on \( \lambda \)-calculus, all partial evaluation rules for this calculus come for free. Furthermore, the optimisations defined in the literature for let-binding can also be used. Figure 8a shows this set of rules.

As the vector constructs of \( \tilde{F} \) are based on pull arrays, one can use the pull-array fusion rules for removing unnecessary intermediate vectors and matrices. The two fusion rules for pull-arrays are shown in Figure 8c. Apart from fusing a pipeline of vector/matrix operations, this optimisation can also be used to derive several matrix-algebraic identities.
Example 5. It is known that for a matrix $M$, the following equality holds \((M^T)^T = M\). We show how we can derive the same equality in $\tilde{\mathcal{F}}$. In other words, we show that:

\[
\text{matrixTranspose (matrixTranspose M) = M}
\]

After let binding the inner expression, and inlining the definition of matrixTranspose and the functions inside it, the following program is produced:

```
(fold f z 0) (fold (fun a i -> f a (i+1)) (fold f z 0) (n - 1)) (fold (fun a i -> let a = z in let i = e0 in f (if e0 then e1 else e2) (if (i = e0) then e1 else e2) else f (e0) else f (e2)) z n)
```

Fig. 8. Transformation Rules for $\tilde{\mathcal{F}}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.
let MT =
    build (length M[0]) (fun i ->
        build (length M) (fun j ->
            M[j][i] ) ) in
build (length MT[0]) (fun i ->
    build (length MT) (fun j ->
        MT[j][i] ) )

Now, by applying the loop fusion rules (cf. Figure 8c) and performing further partial evaluation, the following expression is derived:

build (length M) (fun i ->
    build (length M[0]) (fun j ->
        M[i][j] ) )

This is the same expression as M.

Figure 8d shows the rewrite rules for conditional expressions. The first two rules partially evaluate a conditional expression when its condition is statically known. The next rule removes a conditional expression when both the branches are the same expression. The fourth rewrite rule propagates the result of evaluating the condition into both branches. Finally, the last rewrite rule pushes a function applied to a conditional expression into both branches. This results in duplicating that function, which can lead to explosion in the size of expressions.

Figure 8e corresponds to normalisation rules for the ifold construct. The first two rewrite rules are quite well-known; they unfold a loop the size of which is statically known. The last two rewrite rules are more interesting and can result in asymptotic performance improvements. The third rule turns a loop that does not modify its state into a single statement corresponding to its initial state. The last rule turns a loop that modifies its state only in one of its iterations into a single statement. These two rules are especially useful in the context of dealing with sparse vectors and matrices (e.g., one-hot encoding vectors which can be the result of gradient computations, as well as identity matrices) as we can see in the next example.

**Example 6.** It is known that for a vector v, the following equality holds: \( v \times I = v \), where I is an identity matrix of the same dimension as v. We show how d\( \bar{F} \) derives the same algebraic identity. More specifically, we show that:

```
let I = matrixEye (length v) in
build (length v) (fun i ->
    ifold (length v) 0 (fun a j ->
        a + v[j] * I[j][i] )
)
```

is equivalent to v. Inlining and fusing this expression results in:

```
build (length v) (fun i ->
    ifold (length v) 0 (fun a j ->
        a + v[j] * (if (i=j) then 1 else 0) )
)
```

Applying conditional rules (cf. Figure 8d) and ring-structure rules (cf. Figure 8b) results in:

```
build (length v) (fun i ->
    ifold (length v) 0 (fun a j ->
        if (i=j) then a + v[j] else a )
)
```

After applying the loop normalisation rules (cf. Figure 8e), the following expression is derived:
build (length v) (fun i -> let a = 0 in let j = i in a + v[j])

Finally, performing partial evaluation and simplifications results in the following expression:

build (length v) (fun i -> v[i])

This is the same expression as v.

Let us focus on transformations which are more related to differentiation (but not specific to them). Many intermediate tuples resulting from the dual number technique of AD can be removed by using partial evaluation. Figure 8f shows the partial evaluation rules for removing the intermediate tuples which are followed by a projection.

Partially evaluating the tuples across the boundary of a loop requires a sophisticated analysis of the body of the loop. To simplify this task, we perform loop fission for the loops that return a tuple of values. This is possible only when different elements of the tuple are computed independently in different iterations of the loop. Figure 8g shows how loop fission turns an iteration creating a pair of elements into a pair of two iterations constructing independently the elements of that pair. After performing this optimisation, if we are interested only in a particular element of the result tuple, other loops corresponding to irrelevant elements are removed by partial evaluation.

The next example, shows how \(df\) can derive a well-known algebraic identity for the derivative of matrices by using a sequence of transformation rules defined in this section.

**Example 7.** Based on matrix calculus derivative rules, it is known that \(\frac{\partial}{\partial v_1} (v_1 \cdot v_2) = v_2\), where \(\cdot\) is the vector dot product operator. We would like to show how \(df\) can deduce the same algebraic identity. In other words:

\[
\text{vectorMap (deriv (vectorDot v1 v2) v1) snd} = v2
\]

After expanding the `deriv` macro, \(df\) produces the following program:

```plaintext
vectorMap (build (length v1) (fun i ->
    \(\mathcal{D}[\text{fun v1 v2 -> vectorDot v1 v2}]
    (vectorZip v1 (vectorHot (length v1) i))
    (vectorZip v2 (vectorZeros (length v2)))))
) snd
```

After applying AD transformation rules (cf. Figure 7), the following program is derived:

```plaintext
vectorMap (build (length v1) (fun i ->
    fun v1 v2 -> vectorDot v1 v2
    (vectorZip v1 (vectorHot (length v1) i))
    (vectorZip v2 (vectorZeros (length v2))))
) snd
```

After inlining the definition of `vectorDot` (which is derived by applying the AD transformation rules over the library given in Figure 4), `vectorZip`, `vectorHot`, and `vectorZeros`, and applying the fusion and partial evaluation rules (cf. Figure 8), we have:
After further applying β-reduction (cf. Figure 8a), tuple partial evaluation (cf. Figure 8f), and loop fusion the following program is generated:

\[
\text{build} \left( \text{length } v_1 \right) \left( \text{fun } i \rightarrow \text{snd} \left( \text{ifold} \left( \text{fun } j \rightarrow \begin{array}{l}
(fst \ s) + (fst \ v_1[j]) \ast (fst \ v_2[j]), \\
(snd \ s) + (fst \ v_1[j]) \ast (snd \ v_2[j]) + (snd \ v_1[j]) \ast (fst \ v_2[j])
\end{array} \right) \right) (0, 0) \text{ (length } v_1)\right) \\
\left( \text{build} \left( \text{length } v_1 \right) \left( \text{fun } j \rightarrow (v_1[j], \text{if }(i=j) \text{ then } 1 \text{ else } 0) \right) \right) \\
\left( \text{build} \left( \text{length } v_2 \right) \left( \text{fun } j \rightarrow (v_2[j], 0) \right) \right)
\]

Now we apply loop fission (cf. Figure 8g), conditional rules (cf. Figure 8d), and several other simplification rules:

\[
\text{build} \left( \text{length } v_1 \right) \left( \text{fun } i \rightarrow \text{snd} \left( \text{ifold} \left( \text{fun } s j \rightarrow \begin{array}{l}
(fst \ s) + v_1[j] \ast v_2[j], \\
(snd \ s) + v_1[j] \ast 0 + (\text{if } (i=j) \text{ then } 1 \text{ else } 0) \ast v_2[j]
\end{array} \right) \right) 0 \text{ (length } v_1)\right) \\
\begin{array}{l}
(\text{ifold} \left( \text{fun } s j \rightarrow s + v_1[j] \ast v_2[j] \right) 0 \text{ (length } v_1) \\
(\text{ifold} \left( \text{fun } s j \rightarrow \text{if } (i=j) \text{ then } s + v_2[j] \text{ else } s \right) 0 \text{ (length } v_1)
\end{array}
\]

Note that applying the loop fission rule, does not necessarily improve the performance; it is only after performing tuple partial evaluation rules that the iteration responsible for the original computation is removed and the performance is improved. Thus, the strategy for applying rewrite rules can become tricky. For this particular rewrite rule, we only apply it, when subsequently we can use partial evaluation to further simplify the program. To do so, we define a compound rewrite rule that either applies these rules together, or does not do anything. This has a similar effect to the fold-fusion law, which can be found in the FP literature [Gibbons 2006; Hutton 1999]. After applying the partial evaluation rule, the following program is derived:

\[
\text{build} \left( \text{length } v_1 \right) \left( \text{fun } i \rightarrow (\text{ifold} \left( \text{fun } s j \rightarrow s + v_1[j] \ast v_2[j] \right) 0 \text{ (length } v_1) \\
(\text{if}(i = j) \text{ then } (s + v_2[j]) \text{ else } s) 0 \text{ (length } v_1))\right)
\]

By using the optimisation that turns single access iterations into a single statement (cf. Figure 8e), \(d\overline{f}\) produces the following program:

\[
\text{build} \left( \text{length } v_1 \right) \left( \text{fun } i \rightarrow v_2[i] \right)
\]

This program is equivalent to \(v_2\) if the size of the two input vectors are the same (i.e., \(\text{length } v_1 = \text{length } v_2\)). Otherwise, the input program is ill-formed.

Based on the same set of transformation rules, \(d\overline{f}\) derives other matrix-calculus identities for the gradient of matrices such as \(\frac{\partial \text{tr} (M)}{\partial M} = I\), which states that the derivative of the trace of a matrix

\[\Delta\]
with respect to that matrix, is an identity matrix. More generally, \( \frac{\partial \text{tr}(AM)}{\partial M} = A^T \).

Now we return to the example shown in the beginning of this paper.

**Example 1 (Continued).** If we have a matrix \( M \) and two vectors \( u \) and \( v \) (which are represented as row matrices and are independent of \( M \)), using matrix calculus one can prove that \( \frac{\partial (uMv^T)}{\partial M} = u^Tv \).

First, we start by a partially inlined representation of this program in \( \bar{F} \):

```plaintext
let f = fun u M v ->
  let m =
    matrixMult
      (build 1 (fun i -> u))
      (matrixMult M
        (matrixTranspose (build 1 (fun i -> v))))
  m[0][0]
fun u M v ->
  (build (length M) (fun i ->
    (build (length M[0]) (fun j ->
      (snd (D[f])
        (vectorZip v (vectorZeros (length v))))
        (matrixZip M (matrixHot (length M) (length M[0]) i j))))
        (vectorZip v (vectorZeros (length v))))))
```

Note that the function \( f \) is returning the only scalar element of the 1-by-1 matrix \( uMv^T \). After performing loop fusion, loop fission and partial evaluation the following program is derived:

```plaintext
fun u M v ->
  build (length M) (fun i ->
    build (length M[0]) (fun j ->
      u[i] * v[j]))
```

This program is equivalent to \( u^Tv \) if the input program is well formed, i.e., the number of rows and columns of \( M \) are the same as the length of \( u \) and \( v \), respectively.

△

### 5.2 Code Generation

After applying the optimisations mentioned in the previous section, one can further improve the efficiency by generating programs in a low-level language with manual memory management. This way, the overhead of garbage collection can be removed. Furthermore, by using stack-discipline memory management techniques such as Destination-Passing Style (DPS) [Shaikhha et al. 2017], one can benefit from efficient bump memory allocation instead of using the expensive `malloc` and `free` calls.

**Example 1 (Continued).** The generated C code for the optimised differentiated program is as follows:

```c
matrix uMv_d(storage s, vector u, matrix M, vector v) {
  matrix res = (matrix)s;
  for(int r = 0; r < M->rows; r++) {
    for(int c = 0; c < M->cols; c++) {
      res->elems[r][c] = u->elems[r] * v->elems[c];
    }
  }
}
```
The parameter $s$ is the storage area allocated for storing the result matrix.

Up to now, we have only seen the cases where only the derivative part of the program was of interest. If we are interested in the original part of the program as well (e.g., the intermediate vectors cannot be fused), we need to store both the original and derivative parts. In such cases, the differentiated vectors, which are represented as arrays of tuples, can be transformed into a more efficient data layout. The well-known array of structs (AoS) to struct of arrays (SoA) transformation represents differentiated vectors as a tuple of two numeric arrays. Further partial evaluation can remove the unnecessary decoupled numeric arrays.

5.3 Discussion

As we have seen in the examples of this section, $\tilde{d}$ managed to recover matrix-level algebraic identities (which are normally encoded as high-level optimisations at the level of $\tilde{M}$), by using lower-level rewrite rules in the level of $\tilde{F}$. This is achieved using the 1) algebraic identities available in the scalar arithmetic level (e.g., associativity, commutativity, and distributivity as shown in Figure 8b), 2) $\lambda$-calculus and its extended constructs rewrite rules (as shown in Figures 8a, 8f, and 8d), and 3) the functional loop transformations (e.g., pull-array loop fusion, induction-based loop normalisation rules of $\text{ifold}$, and loop fission as shown in Figures 8c, 8e, and 8g).

Furthermore, we have used these rewrite rules in order to make the differentiation of the vector and matrix expressions more efficient. Even though, we do not have any guarantee that for all programs these sets of rewrite rules are sufficient to make forward-mode AD as efficient as reverse-mode AD, we show how well our technique works in practice in the next section. We achieve these results by relying on the restrictions imposed by the $\tilde{F}$ language, such as only allowing a limited form of recursion (using $\text{ifold}$ and $\text{build}$, thus benefiting from their associated optimisations). Furthermore, these restrictions enable us to produce efficient low-level C code using DPS [Shaikhha et al. 2017].

One of the key challenges for applying these rewrite rules is the order in which these rules should be applied. We apply these rules based on heuristics and cost models for the size of the code (which is used by many optimising compilers, especially the ones for just-in-time scenarios). Furthermore, based on heuristics, we ensure that certain rules are applied only when some specific other rules are applicable. For example, the loop fission rule (Figure 8g) is usually applicable only when it can be combined with tuple projection partial evaluation rules (Figure 8f). We leave the use of search strategies for automated rewriting (e.g., using Monte-Carlo tree search [De Mesmay et al. 2009]) as future work.

6 EXPERIMENTAL RESULTS

In this section, we show how $\tilde{d}$ performs in practice. We show the performance of the differentiated code for micro benchmarks as well as two real-world machine learning and computer vision applications.

Experimental Setup. We have performed the experiments using an iMac machine equipped with an Intel Core i5 CPU running at 2.7GHz, 32GB of DDR3 RAM at 1333Mhz. The operating system is OS X 10.13.1. We use CLang 900.0.39.2 for compiling the generated C code, and Python 2.7.12 for running the Python code, and Mono 5.8.1 for compiling and running F# programs. Furthermore, we use DiffSharp 0.6.3, Tapenade 3.14 (its web interface), Theano 0.9.0, TensorFlow 1.13, and Futhark 0.10.2. For all the experiments, we compute the mean time of ten runs, and the time out is set to
twenty minutes. For the TensorFlow experiments, we do not use the XLA backend and we do not consider the run time of the first round in order to remove the overhead of its graph compilation.

Throughout this section, we compare the performance of the following alternatives:

- **DiffSharp (R)**: The reverse-mode AD of the DiffSharp library [Baydin et al. 2015a].
- **DiffSharp (F)**: The forward-mode AD of the DiffSharp library.
- **Tapenade (R)**: The reverse-mode AD of the Tapenade framework [Hascoet and Pascual 2013].
- **Tapenade (F)**: The forward-mode AD of the Tapenade framework.
- **Theano**: The AD offered by the Theano framework [Bergstra et al. 2010], which is a combination of the reverse-mode AD with symbolic differentiation.
- **TensorFlow**: The AD offered by the Tensorflow framework [Henriksen et al. 2017], which is based on the reverse-mode AD.
- **Futhark**: A forward-mode AD implementation on top of the Futhark programming language [Henriksen et al. 2017].
- **d\tilde{f}**: The code generated by d\tilde{f} with different sets of optimisations (e.g., loop fusion (LF), loop-invariant code motion (LICM), loop normalisation (LN), and Destination-Passing Style (DPS) [Shaikhha et al. 2017] for stack-discipline memory management).

**Micro Benchmarks** which consist of the following vector expressions: 1) gradient of dot product of two vectors with respect to the first vector (which is a Jacobian matrix with a single row), 2) gradient of the maximum value of a vector with respect to the input vector (which is a Jacobian matrix with a single row), 3) gradient of addition of two vectors with respect to the first vector (which is a Jacobian matrix), and 4) gradient of the multiplication of a vector with a scalar value with respect to the scalar value (which is a Jacobian matrix with a single column).

Figure 9 shows the performance results for the mentioned micro benchmarks. In all cases, DiffSharp, Theano, and TensorFlow have performance overhead, which is reduced for larger data sizes thanks to the less interpretation overhead. The Futhark compiler generates C code after applying various types of optimisations such as loop fusion, code motion, dead-code elimination, double buffering (for loops), and standard optimisations like copy-propagation, constant-folding, and CSE among many other optimisations. In all the experiments, we have applied the best of possible optimisations to the programs generated by d\tilde{f} (denoted by d\tilde{f} (Opt)). The performance of these programs is improved further when the generated C code uses DPS for stack-discipline memory management (denoted by d\tilde{f} (Opt) + DPS).

As in the first two cases the Jacobian matrix is a row vector, reverse-mode AD computes the whole Jacobian matrix in a single backward pass. However, forward-mode AD needs to iterate over each column to compute the corresponding derivative value. Hence, Tapenade (F) and DiffSharp (F) have asymptotic performance difference. Even though, the Futhark compiler implements many optimisations mentioned above, the forward-mode AD is asymptotically worse than the reverse-mode AD. On the other hand, DiffSharp (R), Theano, and TensorFlow show performance overhead which is reduced for larger data sizes. In addition, Tapenade (R) and the code generated by d\tilde{f} (d\tilde{f} (Opt)) show similar performance. This shows that the optimisations explained in Section 5 have successfully made the forward-mode AD code of d\tilde{f} as efficient as the reverse-mode AD code. Finally, the performance of d\tilde{f} is improved further when the generated C code uses DPS for stack-discipline memory management.

For the case of the addition of two vectors, as the Jacobian matrix is a square matrix, reverse-mode AD and forward-mode AD show comparable performance. Both AD modes of DiffSharp are from two to three orders of magnitude slower than d\tilde{f}. Both Theano and TensorFlow have performance overhead, which again reduces for bigger data sizes. Specifically, TensorFlow becomes as good as Tapenade (R), Futhark, and d\tilde{f}. 

Finally, for the last case, as the Jacobian matrix is a column vector, the forward mode AD computes the whole Jacobian matrix in a single forward pass. However, the reverse mode AD requires traversing over each row to compute the corresponding partial derivative values. Hence, as opposed to computing the gradient for the dot product and the maximum element of a vector, the systems based on the forward-mode AD show asymptotically better performance.

**Non-Negative Matrix Factorization (NMMF)** is a useful tool which has many applications in various fields ranging from document clustering, recommendation systems, signal processing, to computer vision. For instance, in [Liu et al. 2010], the authors study the NMMF of Web dyadic data represented as the matrix $A$. Dyadic data contains rich information about the interactions between the two participating sets. It is useful for a broad range of practical applications including Web search, Internet monetization, and social media content [Liu et al. 2010]. For example the (query, clicked URL) data is used in query clustering [Ji-rong Wen 2002], query suggestions [Baeza-Yates et al. 2004] and improving search relevance [Agichtein et al. 2006]. Matrix factorization is a commonly used approach to understanding the latent structure of the observed matrix for various applications [Berry et al. 2006; Sra and Dhillon 2006]. The authors present a probabilistic NMMF framework for a variety of Web dyadic data that conforms to different probabilistic distributions. For instance, an Exponential distribution is used to model Web lifetime dyadic data, e.g., user dwell time, and similarly the Poisson distribution is used to model count dyadic data, e.g., click counts.
The iterative algorithm to find $W$ and $H$ depends on the form of the assumed underlying distribution. In particular the update formula for gradient descent are derived by computing the gradient of the negative log of the likelihood function. For example, the negative log of the exponential distribution is represented as follows:

$$D(A||\tilde{A}) = \sum_{(i,j)} \left( \log(\tilde{A}_{i,j}) + \frac{A_{i,j}}{\tilde{A}_{i,j}} \right), \quad \tilde{A} = WH$$

The update formulas are derived manually, and for each new distribution it is the responsibility of the user to undertake the error prone and laborious task of deriving, optimizing, and implementing the update rules. $d\tilde{f}$ automatically derives the gradient of the negative log of the likelihood function for the exponential distribution. After performing optimizations, $d\tilde{f}$ produces an expression which is equivalent to the following update formula, which is manually derived by hand in [Liu et al. 2010]:

$$\frac{\partial D}{\partial H} = W^T \left( \frac{1}{WH} - \frac{A}{(WH)^2} \right)$$
Figure 10 shows the performance results of executing the derived update rule on DiffSharp, Tapenade, Theano, and dF. For all the experiments, we consider factorizing the matrix \( A \) into two vectors \( W \) and \( H \) (represented as \( u \) and \( v^T \), respectively). Comparing Tapenade and dF, we observe that the reverse-mode AD of Tapenade behaves similarly to dF. This shows that dF successfully generates efficient code for this case, which is an ideal case for the reverse-mode AD (the loss function is a scalar valued function, which should compute the gradient with respect to all elements of the input vector). Finally, as the dimension of the vectors increases, Theano and TensorFlow converge to the same performance as dF and reverse-mode AD of Tapenade, because of two reasons. First, the overhead of invoking C functions from Python becomes negligible as the size of the vector increases. Second, Theano and TensorFlow invoke BLAS routines which are highly tuned and vectorised implementations for vector operations.

By comparing different configurations of dF, we observe the following three points. First, the loop fusion improves the performance by around one order of magnitude. The generated C code after applying these optimisations is as efficient as the code generated by Futhark. Second, the loop normalisation (\( LN \)) optimisations (which are the ones shown in Figure 8e) have the most impact by asymptotically improving the performance from two to three orders of magnitude. As explained in Section 5, these transformation rules should be combined with loop fission to become effective. Finally, the DPS representation slightly improves the performance by using the stack-allocation discipline for memory management.

**Bundle Adjustment** [Agarwal et al. 2010; Triggs et al. 1999; Zach 2014] is a computer vision problem, where the goal is to optimise several parameters in order to have an accurate estimate of the projection of a 3D point by a camera. This is achieved by minimizing an objective function representing the reprojection error.

For the experiments, we compute the Jacobian matrix of the Project function in Bundle Adjustment. For a 3D point \( X \in \mathbb{R}^3 \) and a camera with rotation parameter \( r \in \mathbb{R}^3 \), center position \( C \in \mathbb{R}^3 \), focal index \( f \in \mathbb{R} \), principal point \( x_0 \in \mathbb{R}^2 \), and radical distortion \( k \in \mathbb{R}^2 \), the Project function computes the projected point as follows:

\[
\text{project}(r, C, f, x_0, k, X) = \text{distort}(k, p2e(\text{rodrigues}(r, X - C))) f + x_0
\]

\[
\text{distort}(k, x) = x (1 + k_1 ||x||^2 + k_2 ||x||^4)
\]

\[
p2e(X) = X_1,2 \div X_3
\]

\[
\text{rodrigues}(r, X) = X \cos(|r|) + (v \times X) \sin(|r|) + v(v^T X) (1 - \cos(|r|))
\]

\[
\theta = ||r||, v = \frac{r}{||r||}
\]

In order to better demonstrate the expressibility and conciseness of \( \tilde{M} \), Figure 12 shows the implementation of these functions in this language.

Consider having \( N \) 3D points and a vector of 11 camera parameters. We are interested in computing a Jacobian matrix with \( 2N \) rows (the project function produces a 2D output for each input data point) and 11 columns.

Figure 11 shows the performance results for computing the mentioned Jacobian matrix. For this application, as the Jacobian matrix has more rows than columns, the forward-mode AD outperforms the reverse-mode. Thus, we observe that the forward-mode AD of all systems outperforms their reverse-mode. dF outperforms all its competitors by up to three orders of magnitude. More specifically, dF outperforms both forward and reverse mode of Tapenade. We observe that as opposed to the NNMF application, the loop normalisation rules have a negligible impact for this application. However, the loop-invariant code motion optimisations result in up to two times performance improvement by hoisting the shared computation outside the loop. In addition, DPS representation leads to an additional 10% performance improvement. Finally, Futhark outperforms the generated
let distort = fun (radical: Vector) (proj: Vector) ->
    let rsq = vectorNorm proj
    let L = 1.0 + radical.[0] * rsq + radical.[1] * rsq * rsq
    vectorSMul proj L
let rodrigues = fun (rotation: Vector) (x: Vector) ->
    let sqtheta = vectorNorm rotation
    let theta = sqrt sqtheta
    let thetaInv = 1.0 / theta
    let w = vectorSMul rotation thetaInv
    let wCrossX = vectorCross w x
    let tmp = (vectorDot w x) * (1.0 - (cos theta))
    let v1 = vectorSMul x (cos theta)
    let v2 = vectorSMul wCrossX (sin theta)
    vectorAdd (vectorAdd v1 v2) (vectorSMul tmp)
let project = fun (cam: Vector) (x: Vector) ->
    let Xcam = rodrigues (vectorSlice cam 0 2) (vectorSub x (vectorSlice cam 3 5))
    let distorted = distort (vectorSlice cam 9 10) (vectorSMul (vectorSlice Xcam 0 1) (1.0/Xcam.[2]))
    vectorAdd (vectorSlice cam 7 8) (vectorSMul distorted cam.[6])

Fig. 12. Bundle Adjustment functions in \( \tilde{M} \).

C code of \( \tilde{dF} \) mainly thanks to a better data layout representation for tensors, which we plan to integrate into \( \tilde{dF} \) in the future.

7 RELATED WORK

Automatic Differentiation. There is a large body of work on automatic differentiation (AD) of imperative programming languages. Tapenade [Hascoet and Pascual 2013] performs AD for a subset of C and Fortran, whereas, ADIFOR [Bischof et al. 1996] performs AD for Fortran programs.
Adept [Hogan 2014] and ADIC [Narayanan et al. 2010] perform automatic differentiation for C++ by using expression templates. However, as we have seen in our experimental results, an AD tool such as Tapenade misses several optimisation opportunities, mainly due to their limited support for loop fusion and loop-invariant code motion.

ADiMat [Bischof et al. 2002], ADiGator [Weinstein and Rao 2016], and Mad [Forth 2006] perform AD for MATLAB programs, whereas MuPAD [Hivert and Thiéry 2004] computes the derivatives using symbolic differentiation. AutoGrad [Maclaurin et al. 2015] performs AD for Python programs that use NumPy library for array manipulation, whereas Theano [Bergstra et al. 2010] uses symbolic differentiation. Tensorflow [Abadi et al. 2016] performs source-to-source reverse-mode AD, and uses advanced heuristics to solve the memory inefficiencies. ForwardDiff [Revels et al. 2016] employs vector forward-mode AD [Khan and Barton 2015] for differentiating Julia programs. This system keeps a vector of derivative values in the dual number instead of only a single derivative value. All these systems miss important optimisation opportunities such as loop fusion and loop-invariant code motion.

DiffSharp [Baydin et al. 2015a] is an AD library implemented in F#. This library provides both forward-mode and reverse-mode AD techniques. As DiffSharp is a library implementation of AD (in contrast to d̃f, which implements AD as source-to-source transformation rules), it cannot support the simplification rules such as loop-invariant code motion, loop fusion, and partial evaluation. Furthermore, d̃f can efficiently manage memory by generating C code using DPS, whereas DiffSharp should rely on the garbage collection provided by the .NET framework for memory management.

Stalingrad [Pearlmutter and Siskind 2008] is an optimising compiler for a dialect of Scheme with a first-class AD operator, with the support for both forward mode and reverse mode of AD. One of the key challenges that Stalingrad addresses is perturbation confusion [Siskind and Pearlmutter 2005], which occurs for computing the derivative of the functions for which the derivatives are already computed, or the cases where we need the computation of nested differentiation [Pearlmutter and Siskind 2007]. We have shown how d̃f solves the perturbation confusion problem using a static approach thanks to the deriv macro (Section 4.3). One of the main advantages of d̃f over Stalingrad is its support for loop transformations such as loop fusion and loop-invariant code motion.

Karczmarczuk [Karczmarczuk 1999] presents a Haskell implementation for both forward and reverse mode AD. Elliott [Elliott 2009] improves this work by giving a more elegant implementation for its forward mode AD. Furthermore, Elliott [Elliott 2018] provides a generalization of AD based on category theory for implementing both forward and reverse-mode AD. These implementations lack the optimisations offered by transformation rules, especially loop transformations.

D* [Guenter 2007] is a symbolic differentiation system, which performs the factorisation of the common product terms in sum-of-product expressions. d̃f also performs a similar idea by performing common-subexpression elimination and loop-invariant code motion in order to eliminate the redundant computation, hence, improving the performance of forward-mode AD. One of the key limitations of D* is that its input programs should be fully loop unrolled. In other words, its differentiation process does not accept programs with loops. It would be interesting to see if d̃f can be used to optimise the computer graphics applications handled by D*.

Tensorflow [Abadi et al. 2016] and Pytorch [Paszke et al. 2017] are machine learning libraries implemented using Python. These systems are mostly based on tensor abstractions and come with a predefined set of efficient combinators for manipulating tensors. Furthermore, they can use compilation (e.g., the XLA [Leary and Wang 2017] backend for Tensorflow and the Glow [Rotem et al. 2018] backend for PyTorch) in order to perform further optimisations. However, these systems are quite restrictive in what constructs are efficiently supported; additional tensor operations are not as efficient as the predefined set of tensor operators.
Lantern [Wang et al. 2018] uses the multi-stage programming [Taha and Sheard 2000] features provided by LMS [Rompf and Odersky 2010] in order to perform AD for numerical programs written in a subset of Scala. A key feature provided by Lantern is supporting reverse-mode AD by using delimited continuations [Danvy and Filinski 1990]. To the best of our knowledge there is no support for loop fusion nor loop normalisation in Lantern. However, there are some imperative forms of fusion implemented for LMS [Rompf et al. 2013] which Lantern can benefit from. Furthermore, some form of loop-invariant code motion should be achieved thanks to the sea-of-node representation [Click 1995] provided by LMS [Rompf and Odersky 2010]. We plan to implement the reverse-mode AD using closures and continuations as implemented in Lantern [Wang et al. 2018] and Stalingrad [Pearlmutter and Siskind 2008].

Pilatus [Shaikhha and Parreaux 2019] is a linear algebra language which is also using pull arrays for implementing vectors and matrices. Pilatus performs optimisations (e.g., loop fusion and algebraic optimisations) by using multi-stage programming and rewriting facilities of Squid [Parreaux et al. 2017a,b]. However, it does not support loop-invariant code motion and loop normalisation. Apart from supporting forward-mode AD, Pilatus also supports graph processing algorithms and logical probabilistic programming, which can be a future direction for d̃f. Array Languages and Fusion. There are many array programming languages in the literature, APL [Iverson 1962] being the pioneer among them. There are functional array languages such as Futhark [Henriksen et al. 2017] and SAC [Grelck and Scholz 2006] with support for fusion.

In array languages fusion can be achieved by using functional arrays known as push and pull arrays [Anker and Svenningsson 2013; Claessen et al. 2012; Svensson and Svenningsson 2014]. A push-array is represented by an effectful function that, given an index and a value, will write the value into the array. A pull-array is represented by the length of the array and a function producing an element for a given index, similar to the build construct in F. Similarly, functional programming languages use shortcut deforestation for fusing lists either by pulling the stream of data [Coutts et al. 2007; Svenningsson 2002] or pushing them [Gill et al. 1993], which are implemented in Haskell using the rewrite rule facilities of GHC [Jones et al. 2001]. Shortcut deforestation can also be implemented as a library using multi-stage programming [Jonnalagedda and Stucki 2015; Kiselyov et al. 2017; Shaikhha et al. 2018]. It would be interesting to see how the techniques presented in this paper can be implemented on top of other functional array programming languages (e.g., by using GHC rewrite rules or multi-stage programming).

Numerical DSLs. There are many DSLs for numerical workloads. These DSLs can be classified in three categories. The first category consists of mainstream programming languages used by data analysts such as MATLAB and R. These languages offer many toolboxes for performing a wide range of tasks, however, from a performance point of view the focus is only on the efficient implementation of the libraries. The second category consists of DSLs such as Lift [Steuwer et al. 2015], Opt [DeVito et al. 2016], Halide [Ragan-Kelley et al. 2013], Diderot [Chiw et al. 2012], and OptiML [Sujeeth et al. 2011], which generate parallel code from their high-level programs. The third category is the DSLs which focus on generating efficient machine code for fixed size linear algebra problems such as Spiral [Puschel et al. 2005] and LGen [Spampinato and Püschel 2016]. These DSLs exploit the memory hierarchy by relying on searching algorithms for making tiling and scheduling decisions. Except the first category, for which automatic differentiation tools exist, the other DSLs do not have any support for automatic differentiation. Moreover, parallel code generation and efficient machine code generation are orthogonal concepts and can be added to d̃f in the future.
8 CONCLUSIONS

In this paper we have demonstrated how to efficiently compute the derivative of a program. The key idea behind our system is exposing all the constructs used in differentiated programs to the underlying compiler. As a result, the compiler can apply various loop transformations such as loop-invariant code motion and loop fusion for optimizing differentiated programs. We have shown how $\text{dF}$ outperforms the existing AD tools on micro benchmarks and real-world machine learning and computer vision applications.

ACKNOWLEDGMENTS

The first author was partially supported by EPFL and a Google Ph.D. Fellowship during the preparation of this paper.

REFERENCES


Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion. From Lists to Streams to Nothing at All (ICFP ’07).


Efficient Differentiable Programming in a Functional Array-Processing Language 97:29

