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**; *Domain specific languages.*

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

ACM Reference Format:
Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (August 2019), 30 pages. https://doi.org/10.1145/3341701

> *... 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.

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]:

$$
\begin{aligned}
&f\left(x_1, ..., x_n\right) = \\
&\quad \texttt{let}\, v_1 = e_1 \\
&\quad ... \\
&\quad \texttt{let}\, v_n = e_n \\
&\quad v_n
\end{aligned}
$$

To compute the derivative of this function using the forward-mode AD, we associate a *tangent* variable to each variable v_i, denoted by $\overrightarrow{v_i}$. Each tangent variable is computed as $\overrightarrow{v_i} = \overrightarrow{x_1} \times \frac{\partial v_i}{\partial x_1} + ... + \overrightarrow{x_n} \times \frac{\partial v_i}{\partial x_n}$. In order to compute the partial derivative of f with respect to x_i, we assign $\overrightarrow{x_i} = 1$ and $\overrightarrow{x_j} = 0$ 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 $\overleftarrow{v_i}$, which is computed as $\overleftarrow{v_i} = \frac{\partial y}{\partial v_i}$. 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 $\overleftarrow{x_1}$. Similarly, if we are interested in the partial derivative of this function with respect

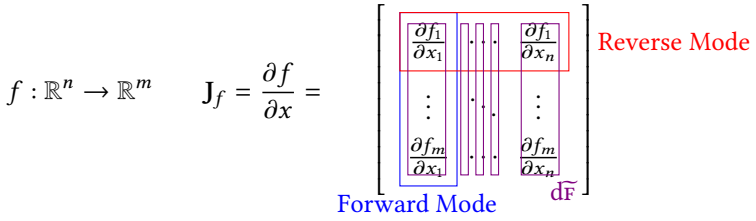

Fig. 1. The Jacobian Matrix of a Function. Forward-mode AD computes a column of this matrix, whereas the reverse-mode AD computes a row of this matrix. $\widetilde{\mathrm{dF}}$ computes the full Jacobian matrix using a vectorized variant of the forward-mode AD.

to x_2, we have to compute the value of $\overleftarrow{x_2}$. 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 $\mathbf{J}$ 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 $\widetilde{\mathrm{dF}}$, which is shown in Figure 2. This figure demonstrates the position of $\widetilde{\mathrm{dF}}$ with respect to existing AD tools.

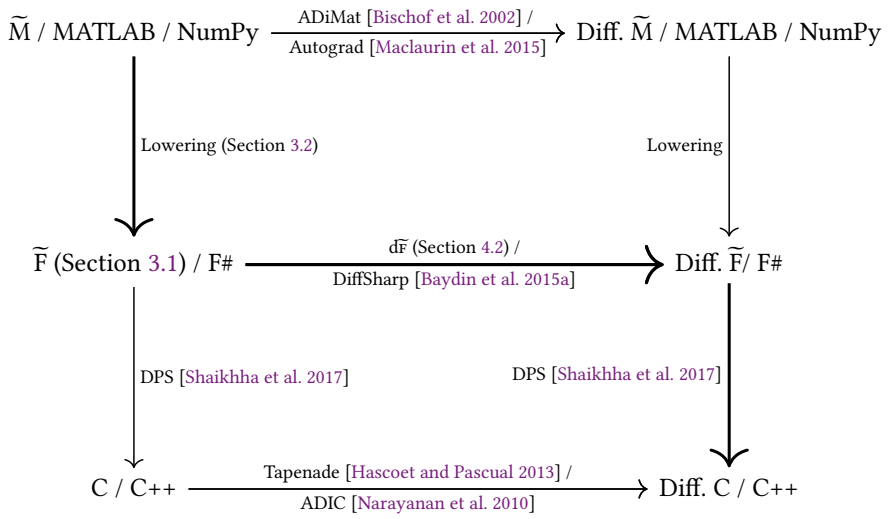

Fig. 2. Compilation process in different AD systems. The solid arrows correspond to the pipeline used in dF.

dF starts from a program written in a high-level linear algebra library, called M (Section 3.2). This program is lowered into its implementation in a higher-order functional language with array support, called F (Section 3.1). If a part of the program requires computing differentiation (which are specified by using high-level differentiation API exposed by dF, as mentioned in Section 4.1) dF 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 F programs.

Next, we present the core functional language used in dF, on top of which we define source-to-source AD transformation and simplification rules.

3.1 F

F (pronounced as F smooth) is a subset of 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 F. $\overline{x}$ and $\overline{e}$ denote one or more variables and expressions, respectively, which are separated by spaces, whereas, $\overline{T}$ represents one or more types which are separated by $\Rightarrow$. In addition to the usual λ-calculus constructs (abstraction, application, and variable access), F supports let binding and conditionals.

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

Typing Rules:

$$\text{(T-App)}\ \frac{e_0 : \overline{T} \Rightarrow M \qquad \overline{e} : \overline{T}}{e_0\ \overline{e} : M} \qquad \text{(T-Abs)}\ \frac{\Gamma \cup \overline{x} : \overline{T} \vdash e : M}{\Gamma \vdash \lambda \overline{x}.e : \overline{T} \Rightarrow M} \qquad \text{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2} \qquad \text{(T-If)}\ \frac{e_1 : \texttt{Bool} \qquad e_2 : M \qquad e_3 : M}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : M}$$

Scalar Function Constants:

`+ | - | * | / | **`	:	Num, Num ⇒ Num	`> | < | == | <>`	:	Num ⇒ Num ⇒ `Bool`
`sin | cos | tan |`			`&& | ||`	:	`Bool` ⇒ `Bool` ⇒ `Bool`
`log | exp`	:	Num ⇒ Num	`!`	:	`Bool` ⇒ `Bool`

Vector Function Constants:

`build` : `Card` ⇒ (`Index` ⇒ M) ⇒ `Array<M>` `get` : `Array<M>` ⇒ `Index` ⇒ M

`ifold` : (M ⇒ `Index` ⇒ M) ⇒ M ⇒ `Card` ⇒ M `length` : `Array<M>` ⇒ `Card`

Pair Function Constants:

`pair` : $M_1 \Rightarrow M_2 \Rightarrow M_1 \times M_2$ `fst` : $M_1 \times M_2 \Rightarrow M_1$ `snd` : $M_1 \times M_2 \Rightarrow M_2$

Syntactic Sugar:

$e_0[e_1]$	=	`get` e_0 e_1	`Matrix`	=	`Array<Array<Double>>`
(e_0, e_1)	=	`pair` e_0 e_1	`DoubleD`	=	`Double` × `Double`
e_1 *bop* e_2	=	*bop* e_1 e_2	`VectorD`	=	`Array<Double` × `Double>`
`Vector`	=	`Array<Double>`	`MatrixD`	=	`Array<Array<Double` × `Double>>`

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

$\widetilde{\text{F}}$ 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 $\widetilde{\text{F}}$ 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 $\widetilde{\text{F}}$.

Table 1. Equivalent operations in Matlab, R, NumPy, and $\widetilde{\text{M}}$.

Matlab	R	NumPy	$\widetilde{\text{M}}$
A * B	A %*% B	A.dot(B)	matrixMult A B
A + B	A + B	A + B	matrixAdd A B
A'	t(A)	A.T	matrixTranspose A
ones(n, m)	matrix(1, n, m)	ones((n, m))	matrixOnes n m
zeros(n, m)	matrix(0, n, m)	zeros((n, m))	matrixZeros n m
eye(n)	diag(n)	eye(n)	matrixEye n

3.2 $\widetilde{\text{M}}$

$\widetilde{\text{M}}$ 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, $\widetilde{\text{M}}$ frees the users from low-level details and enables them to focus on the algorithmic aspects of the problem in hand.

$\widetilde{\text{M}}$ is simply a $\widetilde{\text{F}}$ library, but it can also be thought of as an *embedded domain-specific language* (EDSL) [Hudak 1996]. Figure 4 demonstrates a subset of $\widetilde{\text{M}}$ operations which are defined as functions in $\widetilde{\text{F}}$. 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, $\widetilde{\text{M}}$ is inspired by MATLAB and R. As a result, there is a mapping among the constructs of $\widetilde{\text{M}}$ and these matrix-based languages. Hence, it is easily possible to translate a program written in one of these languages to $\widetilde{\text{M}}$. Table 1 demonstrates the mapping among a subset of the constructs of MATLAB, R, NumPy and $\widetilde{\text{M}}$.

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 $\frac{\partial\left(uMv^T\right)}{\partial M} = u^T v$. 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 $\text{d}\widetilde{\text{F}}$ derives the gradient of this expression with respect to M, resulting in an expression equivalent to $u^T v$. 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 $\widetilde{\text{M}}$:

```
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 1×1 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 $\widetilde{\text{F}}$ [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.

```
let vectorRange = fun n ->
  build n (fun i -> i)
let vectorFill = fun n e ->
  build n (fun i -> e)
let vectorHot = fun n i ->
  build n (fun j -> if i = j then 1 else 0)
let vectorMap = fun v f ->
  build (length v) (fun i -> f v[i])
let vectorMap2 = fun v1 v2 f ->
  build (length v1) (fun i -> f v1[i] v2[i])
let vectorZip = fun v1 v2 ->
  vectorMap2 v1 v2 (pair)
let vectorAdd = fun v1 v2 ->
  vectorMap2 v1 v2 (+)
let vectorEMul = fun v1 v2 ->
  vectorMap2 v1 v2 (×)
let vectorSMul = fun v s ->
  vectorMap v (fun a -> a × s)
let vectorSum = fun v ->
  ifold (fun s i -> s + v[i]) 0 (length v)
let vectorDot = fun v1 v2 ->
  vectorSum (vectorEMul v1 v2)
let vectorNorm = fun v ->
  sqrt (vectorDot v v)
let vectorSlice = fun v s e ->
  build (e − s + 1) (fun i -> v[i + s])
let vectorToMatrix = fun v ->
  build 1 (fun i -> v)
let vectorOutProd = fun v1 v2 ->
  let m1 = vectorToMatrix v1
  let m2 = vectorToMatrix v2
  let m2T = matrixTranspose m2
  matrixMul m1 m2T
```

```
let matrixRows = fun m -> length m
let matrixCols = fun m -> length (m[0])
let matrixZeros = fun r c ->
  build r (fun i -> vectorFill c 0)
let matrixOnes = fun r c ->
  build r (fun i -> vectorFill c 1)
let matrixEye = fun n ->
  build n (fun i -> vectorHot n i)
let matrixHot = fun n m r c ->
  build n (fun i ->
    build m (fun j ->
      if (i = r && j = c) then 1 else 0
  ) )
let matrixMap = fun m f ->
  build (length m) (fun i -> f m[i])
let matrixMap2 = fun m1 m2 f ->
  build (length m1) (fun i -> f m1[i] m2[i])
let matrixAdd = fun m1 m2 ->
  matrixMap2 m1 m2 vectorAdd
let matrixTranspose = fun m ->
  build (matrixCols m) (fun i ->
    build (matrixRows m) (fun j ->
      m[j][i]
  ) )
let matrixMul = fun m1 m2 ->
  let m2T = matrixTranspose m2
  build (matrixRows m1) (fun i ->
    build (matrixCols m2) (fun j ->
      vectorDot (m1[i]) (m2T[j])
  ) )
let matrixTrace = fun m ->
  ifold (fun s i -> s+m[i][i]) 0 (length m)
```

Fig. 4. A subset of $\widetilde{\mathrm{M}}$ constructs defined in $\widetilde{\mathrm{F}}$.

4 DIFFERENTIATION

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

4.1 High-Level API

For computing the derivative of an arbitrary function, $\mathrm{d}\widetilde{\mathrm{F}}$ 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 $\mathcal{G}[\![\text{e}]\!]_{\text{x}}$ 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 $\mathcal{D}[\![]\!]$), 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 $\mathcal{A}_i[\![]\!]$ 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:

```
snd (deriv (cos a) a)
```

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

```
snd ((D[[fun a -> cos a]]) (a, 1))
```

△

Furthermore, $\widetilde{\text{dF}}$ provides three additional differentiation constructs, inspired by AD tools such as DiffSharp [Baydin et al. 2015a]: 1) `diff` computes the derivative of a function, from a real number to a real number, with respect to its input, 2) `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) `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 $\mathcal{D}$.

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

```
g = fun x -> cos(x)
```

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

[2] `deriv` 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.

$$
\begin{array}{rl}
\texttt{deriv e x} = & \mathcal{G}[\![\texttt{e}]\!]_{\texttt{x}} \\
\\
\mathcal{G}[\![\texttt{e}]\!]_{\texttt{x}} = & \\
\text{if x: } \texttt{Double} & (\mathcal{D}[\![\texttt{fun } \overline{\texttt{v}_i} \texttt{ -> e}]\!])\ \overline{\mathcal{A}_0[\![\texttt{v}_i]\!]_{\texttt{x}}} \\
\text{if x: } \texttt{Vector} & \texttt{build (length x) (fun r -> } (\mathcal{D}[\![\texttt{fun } \overline{\texttt{v}_i} \texttt{ -> e}]\!])\ \overline{\mathcal{A}_1[\![\texttt{v}_i]\!]_{\texttt{x,r}}}\texttt{)} \\
\text{if x: } \texttt{Matrix} & \texttt{build (matrixRows x) (fun r ->} \\
 & \quad \texttt{build (matrixCols x) (fun c -> } (\mathcal{D}[\![\texttt{fun } \overline{\texttt{v}_i} \texttt{ -> e}]\!])\ \overline{\mathcal{A}_2[\![\texttt{v}_i]\!]_{\texttt{x,r,c}}}\texttt{))} \\
\textbf{where} & \\
\overline{\texttt{v}_i} = & \text{fvs(e)} \\
\\
\mathcal{A}_0[\![\texttt{x}]\!]_{\texttt{x}} = & \text{(x, 1)} \\
\mathcal{A}_0[\![\texttt{v}]\!]_{\texttt{x}} = & \text{(v, 0)} \\
\\
\mathcal{A}_1[\![\texttt{x}]\!]_{\texttt{x,r}} = & \text{vectorZip x (vectorHot (length x) r)} \\
\mathcal{A}_1[\![\texttt{v}]\!]_{\texttt{x,r}} = & \text{vectorZip v (vectorZeros (length v))} \\
\\
\mathcal{A}_2[\![\texttt{x}]\!]_{\texttt{x,r,c}} = & \text{matrixZip x (matrixHot (matrixRows x) (matrixCols x) r c)} \\
\mathcal{A}_2[\![\texttt{v}]\!]_{\texttt{x,r,c}} = & \text{matrixZip v (matrixZeros (matrixRows v) (matrixCols v))}
\end{array}
$$

Fig. 5. Implementation of the deriv construct as a source-to-source transformation pass.

Oper.	Type	Definition
`diff`	(Double⇒Double) ⇒ Double⇒DoubleD	`fun f x -> 𝒟⟦f⟧ (x, 1)`
`grad`	(Vector⇒Double) ⇒Vector⇒VectorD	`fun f v -> build (length v) (fun i -> 𝒟⟦f⟧ (vectorZip v (vectorHot (length v) i)) )`
`jacob`	(Vector⇒Vector) ⇒Vector⇒MatrixD	

Fig. 6. High-Level Differentiation API for $\widetilde{\text{F}}$.

Table 2. Different types of matrix derivatives.

Input Type \ Output Type	Scalar	Vector	Matrix
Scalar	`diff`	`vdiff`	`mdiff`
Vector	`grad`	`jacob`	–
Matrix	`mgrad`	–	–

```
snd ((diff g) a)
```

which is expanded to the following program:

```
snd (𝒟⟦g⟧ (a, 1))
```

△

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.

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

$\widetilde{\text{dF}}$ 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. $\mathcal{D}[\![\text{e}]\!]$ specifies the AD translation for expression e. A variable y is translated as $\vec{\text{y}}$, emphasizing that the translated variable keeps the derivative part as well (D-Abs, D-Var, and D-Let). $\mathcal{P}[\![\text{e}]\!]$ is a shorthand for extracting the original computation from the translated term $\mathcal{D}[\![\text{e}]\!]$, while $\mathcal{E}[\![\text{e}]\!]$ 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:

$\vec{\text{g}}$ = `fun` $\vec{\text{x}}$ `->` `-snd` ($\vec{\text{x}}$) `*` `sin(fst` ($\vec{\text{x}}$)`)`

Based on the definition of the `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:

`snd ((diff g) a)` $\leadsto$ `snd` ($\mathcal{D}[\![\text{g}]\!]$ `(a, 1))` $\leadsto$ `snd` ($\vec{\text{g}}$ `(a, 1))` $\leadsto$

`-snd ((a, 1)) * sin(fst ((a, 1)))` $\leadsto$ `-1 * sin(a)` $\leadsto$ `-sin(a)`

△

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 * b with respect to a, which is represented as follows in $\widetilde{\text{F}}$:

```
snd (deriv (a * b) a)
```

This expression is expanded as follows:

$$\texttt{snd}\ (\mathcal{D}[\![\texttt{fun a b -> a * b}]\!]\ (a, 1)\ (b, 0))$$

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

$$\texttt{snd}\ ((\texttt{fun}\ \vec{a}\ \vec{b}\ \texttt{->}\ (\texttt{fst}\ \vec{(a)} \texttt{*} \texttt{fst}\ \vec{(b)},\ \texttt{fst}\ \vec{(a)} \texttt{*} \texttt{snd}\ \vec{(b)} + \texttt{snd}\ \vec{(a)} \texttt{*} \texttt{fst}\ \vec{(b)}))\ (a, 1)\ (b, 0))$$

which is evaluated as follows:

$$\texttt{snd}\ ((a * b, 1 * b + a * 0)) \quad \rightsquigarrow \quad 1 * b + a * 0 \quad \rightsquigarrow \quad b$$

△

It is important to note that $\widetilde{\mathrm{dF}}$ 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 $\widetilde{\mathrm{dF}}$.

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 $\widetilde{\mathrm{F}}$ 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 $\widetilde{\mathrm{F}}$ language:

```
fun x y ->
  snd (
    deriv (x * (snd (
      deriv (x + y) y
    ))) x
  )
```

After expanding the inner `deriv` macro, the following expression is derived:

$$
\begin{array}{lrcl}
\text{(D-App)} & \mathcal{D}[\![\texttt{e}_0\ \texttt{e}_1]\!] & = & (\mathcal{D}[\![\texttt{e}_0]\!])\ (\mathcal{D}[\![\texttt{e}_1]\!]) \\
\text{(D-Abs)} & \mathcal{D}[\![\texttt{fun x -> e}]\!] & = & \texttt{fun}\ \vec{\texttt{x}}\ \texttt{->}\ \mathcal{D}[\![\texttt{e}]\!] \\
\text{(D-Var)} & \mathcal{D}[\![\texttt{y}]\!] & = & \vec{\texttt{y}} \\
\text{(D-Let)} & \mathcal{D}[\![\texttt{let x = e}_1\ \texttt{in e}_2]\!] & = & \texttt{let}\ \vec{\texttt{x}} = \mathcal{D}[\![\texttt{e}_1]\!]\ \texttt{in} \\
 & & & \quad \mathcal{D}[\![\texttt{e}_2]\!] \\
\text{(D-If)} & \mathcal{D}[\![\texttt{if e}_1\ \texttt{then e}_2\ \texttt{else e}_3]\!] & = & \texttt{if}\ (\texttt{fst}\ \mathcal{D}[\![\texttt{e}_1]\!])\ \texttt{then}\ \mathcal{D}[\![\texttt{e}_2]\!]\ \texttt{else}\ \mathcal{D}[\![\texttt{e}_3]\!] \\
 & & & \\
\text{(D-Build)} & \mathcal{D}[\![\texttt{build e}_0\ \texttt{e}_1]\!] & = & \texttt{build}\ (\texttt{fst}\ \mathcal{D}[\![\texttt{e}_0]\!])\ (\texttt{fun i ->}\ (\mathcal{D}[\![\texttt{e}_1]\!])\ (\texttt{i}, 0)) \\
\text{(D-IFold)} & \mathcal{D}[\![\texttt{ifold e}_0\ \texttt{e}_1\ \texttt{e}_2]\!] & = & \texttt{ifold}\ (\texttt{fun x i ->} \\
 & & & \quad (\mathcal{D}[\![\texttt{e}_0]\!])\ \texttt{x}\ (\texttt{i}, 0))\ \mathcal{D}[\![\texttt{e}_1]\!]\ (\texttt{fst}\ \mathcal{D}[\![\texttt{e}_2]\!]) \\
\text{(D-Get)} & \mathcal{D}[\![\texttt{e}_0[\texttt{e}_1]]\!] & = & (\mathcal{D}[\![\texttt{e}_0]\!])[\texttt{fst}\ \mathcal{D}[\![\texttt{e}_1]\!]] \\
\text{(D-Length)} & \mathcal{D}[\![\texttt{length e}_0]\!] & = & (\texttt{length}\ \mathcal{D}[\![\texttt{e}_0]\!], 0) \\
 & & & \\
\text{(D-Pair)} & \mathcal{D}[\![(\texttt{e}_0, \texttt{e}_1)]\!] & = & (\mathcal{D}[\![\texttt{e}_0]\!], \mathcal{D}[\![\texttt{e}_1]\!]) \\
\text{(D-Fst)} & \mathcal{D}[\![\texttt{fst e}_0]\!] & = & \texttt{fst}\ (\mathcal{D}[\![\texttt{e}_0]\!]) \\
\text{(D-Snd)} & \mathcal{D}[\![\texttt{snd e}_0]\!] & = & \texttt{snd}\ (\mathcal{D}[\![\texttt{e}_0]\!]) \\
 & & & \\
\text{(D-Scalar)} & \mathcal{D}[\![\texttt{e}]\!] & = & (\mathcal{P}[\![\texttt{e}]\!], \mathcal{E}[\![\texttt{e}]\!]) \\
 & & & \\
\text{(D-Add)} & \mathcal{E}[\![\texttt{e}_1 + \texttt{e}_2]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] + \mathcal{E}[\![\texttt{e}_2]\!] \\
\text{(D-Mult)} & \mathcal{E}[\![\texttt{e}_1 * \texttt{e}_2]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] * \mathcal{P}[\![\texttt{e}_2]\!] + \mathcal{P}[\![\texttt{e}_1]\!] * \mathcal{E}[\![\texttt{e}_2]\!] \\
\text{(D-Div)} & \mathcal{E}[\![\texttt{e}_1 / \texttt{e}_2]\!] & = & (\mathcal{E}[\![\texttt{e}_1]\!] * \mathcal{P}[\![\texttt{e}_2]\!] - \mathcal{P}[\![\texttt{e}_1]\!] * \mathcal{E}[\![\texttt{e}_2]\!]) / (\mathcal{P}[\![\texttt{e}_2]\!] ** 2) \\
\text{(D-Neg)} & \mathcal{E}[\![\texttt{-e}_1]\!] & = & \texttt{-}\mathcal{E}[\![\texttt{e}_1]\!] \\
\text{(D-Pow)} & \mathcal{E}[\![\texttt{e}_1 ** \texttt{e}_2]\!] & = & (\mathcal{P}[\![\texttt{e}_2]\!] * \mathcal{E}[\![\texttt{e}_1]\!] / \mathcal{P}[\![\texttt{e}_1]\!] + \log(\mathcal{P}[\![\texttt{e}_1]\!]) * \mathcal{E}[\![\texttt{e}_2]\!]) * \\
 & & & \quad (\mathcal{P}[\![\texttt{e}_1]\!] ** \mathcal{P}[\![\texttt{e}_2]\!]) \\
\text{(D-Sin)} & \mathcal{E}[\![\texttt{sin e}_1]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] * (\cos\ \mathcal{P}[\![\texttt{e}_1]\!]) \\
\text{(D-Cos)} & \mathcal{E}[\![\texttt{cos e}_1]\!] & = & \texttt{-}\mathcal{E}[\![\texttt{e}_1]\!] * (\sin\ \mathcal{P}[\![\texttt{e}_1]\!]) \\
\text{(D-Tan)} & \mathcal{E}[\![\texttt{tan e}_1]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] / ((\cos\ \mathcal{P}[\![\texttt{e}_1]\!]) ** 2) \\
\text{(D-Log)} & \mathcal{E}[\![\texttt{log e}_1]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] / \mathcal{P}[\![\texttt{e}_1]\!] \\
\text{(D-Exp)} & \mathcal{E}[\![\texttt{exp e}_1]\!] & = & \mathcal{E}[\![\texttt{e}_1]\!] * (\exp\ \mathcal{P}[\![\texttt{e}_1]\!]) \\
 & & & \\
\text{(DT-Fun)} & \mathcal{D}_{\mathcal{T}}[\![\texttt{T}_1 \Rightarrow \texttt{T}_2]\!] & = & \mathcal{D}_{\mathcal{T}}[\![\texttt{T}_1]\!] \Rightarrow \mathcal{D}_{\mathcal{T}}[\![\texttt{T}_2]\!] \\
\text{(DT-Exp)} & \mathcal{D}_{\mathcal{T}}[\![\texttt{Num}]\!] & = & \texttt{Num} \times \texttt{Num} \\
\text{(DT-Arr)} & \mathcal{D}_{\mathcal{T}}[\![\texttt{Array<M>}]\!] & = & \texttt{Array<}\mathcal{D}_{\mathcal{T}}[\![\texttt{M}]\!]\texttt{>} \\
\text{(DT-Pair)} & \mathcal{D}_{\mathcal{T}}[\![\texttt{M}_1 \times \texttt{M}_2]\!] & = & \mathcal{D}_{\mathcal{T}}[\![\texttt{M}_1]\!] \times \mathcal{D}_{\mathcal{T}}[\![\texttt{M}_2]\!]
\end{array}
$$

Fig. 7. Automatic Differentiation Rules for $\widetilde{\text{F}}$ Expressions.

```
fun x y ->
  snd (
    deriv (
      let t1 = snd (
        (fun x⃗ y⃗ -> (fst (x⃗) + fst (y⃗), snd (x⃗) + snd (y⃗))) (x, 0) (y, 1)
      )
      x * t1) x
  )
```

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 `deriv` macro results in the following expression:

```
fun x y ->
  snd (
    (fun x⃗ y⃗ ->
      let t⃗1 = snd (
        (fun x⃗⃗ y⃗⃗ ->
          ( ( fst (fst (x⃗⃗)) + fst (fst (y⃗⃗)) ,  snd (fst (x⃗⃗)) + snd (fst (y⃗⃗)) ) ,
            ( fst (snd (x⃗⃗)) + fst (snd (y⃗⃗)) ,  snd (snd (x⃗⃗)) + snd (snd (y⃗⃗)) ) )
        ) (x⃗, (0, 0)) (y⃗, (1, 0))
      )
      (fst (x⃗) * fst (t⃗1), snd (x⃗) * fst (t⃗1) + fst (x⃗) * snd (t⃗1))
    ) (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:

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

△

5 EFFICIENT DIFFERENTIATION

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

5.1 Transformation Rules

There are various algebraic identities that one can define for $\widetilde{\mathrm{F}}$. 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 $\widetilde{\mathrm{F}}$. 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 $\mathrm{d}\widetilde{\mathrm{F}}$ applies. This is because they do not necessarily improve the performance, unless they are combined with other rewrite rules.

As $\widetilde{\mathrm{F}}$ is based on λ-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 $\widetilde{\mathrm{F}}$ 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 pipline of vector/matrix operations, this optimisation can also be used to derive several matrix-algebra identities.

(fun x -> e_0) e_1	$\rightsquigarrow$	let x = e_1 in e_0
let x = e_0 in e_1	$\rightsquigarrow$	$e_1[x \mapsto e_0]$
let x = e_0 in e_1	$\rightsquigarrow$	e_1 ($x \notin fvs(e_1)$)
let x = let y = e_0 in e_1 in e_2	$\rightsquigarrow$	let y = e_0 in let x = e_1 in e_2
let x = e_0 in let y = e_0 in e_1	$\rightsquigarrow$	let x = e_0 in let y = x in e_1
let x = e_0 in let y = e_1 in e_2	$\rightsquigarrow$	let y = e_1 in let x = e_0 in e_2
f(let x = e_0 in e_1)	$\rightsquigarrow$	let x = e_0 in f(e_1)

(a) λ-Calculus Rules

e + 0 = 0 + e	$\rightsquigarrow$	e
e * 1 = 1 * e	$\rightsquigarrow$	e
e * 0 = 0 * e	$\rightsquigarrow$	0
e + -e = e - e	$\rightsquigarrow$	0
e_0 * e_1 + e_0 * e_2	$\rightsquigarrow$	e_0 * (e_1 + e_2)

(b) Ring-Structure Rules

(build e_0 e_1)[e_2]	$\rightsquigarrow$	e_1 e_2
length (build e_0 e_1)	$\rightsquigarrow$	e_0

(c) Loop Fusion Rules

if true then e_1 else e_2	$\rightsquigarrow$	e_1
if false then e_1 else e_2	$\rightsquigarrow$	e_2
if e_0 then e_1 else e_1	$\rightsquigarrow$	e_1
if e_0 then e_1 else e_2	$\rightsquigarrow$	if e_0 then $e_1[e_0 \mapsto$ true] else $e_2[e_0 \mapsto$ false]
f (if e_0 then e_1 else e_2)	$\rightsquigarrow$	if e_0 then f (e_1) else f (e_2)

(d) Conditional Rules

ifold f z 0	$\rightsquigarrow$	z
ifold f z n	$\rightsquigarrow$	ifold (fun a i -> f a (i+1)) (f z 0) (n - 1)
ifold (fun a i -> a) z n	$\rightsquigarrow$	z
ifold (fun a i -> if(i = e_0) then e_1 else a) z n	$\rightsquigarrow$	let a = z in let i = e_0 in e_1 *(if e_0 does not mention a or i)*

(e) Loop Normalisation Rules

fst (e_0, e_1)	$\rightsquigarrow$	e_0
snd (e_0, e_1)	$\rightsquigarrow$	e_1

(f) Tuple Normalisation Rules

ifold (fun a i -> (f_0 (fst a) i, f_1 (snd a) i)) (z_0, z_1) n $\rightsquigarrow$ (ifold f_0 z_0 n, ifold f_1 z_1 n)

(g) Loop Fission Rule

Fig. 8. Transformation Rules for $\widetilde{\mathsf{F}}$. Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

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 $\mathrm{d}\widetilde{\mathsf{F}}$. In other words, we show that:

matrixTranspose (matrixTranspose M) = M

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

```
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 $\overrightarrow{\mathrm{dF}}$ 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 $\widetilde{\mathrm{dF}}$ 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 $\frac{\partial(v_1 \cdot v_2)}{\partial v_1} = v_2$, where $\cdot$ is the vector dot product operator. We would like to show how $\widetilde{\mathrm{dF}}$ can deduce the same algebraic identity. In other words:

```
vectorMap (deriv (vectorDot v1 v2) v1) snd) = v2
```

After expanding the `deriv` macro, $\widetilde{\mathrm{dF}}$ produces the following program:

```
vectorMap (
  build (length v1) (fun i ->
    𝒟⟦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:

```
vectorMap (
  build (length v1) (fun i ->
    fun v1⃗ v2⃗ -> vectorDot⃗ v1⃗ v2⃗
      (vectorZip v1 (vectorHot (length v1) i))
      (vectorZip v2 (vectorZeros (length v2))))
) snd
```

After inlining the definition of $\overrightarrow{\text{vectorDot}}$ (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:

```
build (length v1) (fun i ->
  snd (fun v1⃗ v2⃗ -> ifold (fun s j ->
          ( (fst s) + (fst v1⃗[j]) * (fst v2⃗[j]) ,
            (snd s) + (fst v1⃗[j]) * (snd v2⃗[j]) + (snd v1⃗[j]) * (fst v2⃗[j]) )
          ) (0, 0) (length v1⃗))
        (build (length v1) (fun j -> (v1[j], if(i=j) then 1 else 0)))
        (build (length v2) (fun j -> (v2[j], 0)))))
```

After further applying β-reduction (cf. Figure 8a), tuple partial evaluation (cf. Figure 8f), and loop fusion the following program is generated:

```
build (length v1) (fun i ->
  snd (ifold (fun s j ->
          ( (fst s) + v1[j] * v2[j] ,
            (snd s) + v1[j] * 0 + (if (i=j) then 1 else 0) * v2[j] )
          ) (0, 0) (length v1))
```

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

```
build (length v1) (fun i ->
  snd (
     ifold (fun s j -> s + v1[j] * v2[j]) 0 (length v1) ,
     ifold (fun s j -> if (i=j) then s + v2[j] else s) 0 (length v1)
  ) )
```

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 paritcular 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 iterature [Gibbons 2006; Hutton 1999]. After applying the partial evaluation rule, the following program is derived:

```
build (length v1) (fun i ->
  (ifold (fun s j ->
     if(i = j) then
       (s + v2[j])
     else
       s) 0 (length v1)))
```

By using the optimisation that turns single access iterations into a single statement (cf. Figure 8e), $\widetilde{\mathrm{dF}}$ produces the following program:

```
build (length v1) (fun i -> v2[i])
```

This program is equivalent to $v2$ if the size of the two input vectors are the same (i.e., `length` $v1$ = `length` $v2$). Otherwise, the input program is ill-formed.

△

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

with respect to that matrix, is an identity matrix. More generally, $\widetilde{\mathrm{dF}}$ can automatically discover the following algebraic identity if A is independent of M: $\frac{\partial tr\left(MA\right)}{\partial M} = A^T$.

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 $\frac{\partial\left(uMv^T\right)}{\partial M} = u^T v$. First, we start by a partially inlined representation of this program in $\widetilde{\mathrm{F}}$:

```
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:

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

This program is equivalent to $u^T v$ 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:

```
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];
    }
```

```
    }
    return res;
  }
```

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, $\mathrm{d}\widetilde{\mathrm{F}}$ managed to recover matrix-level algebraic identities (which are normally encoded as high-level optimisations at the level of $\widetilde{\mathrm{M}}$), by using lower-level rewrite rules in the level of $\widetilde{\mathrm{F}}$. 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) λ-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 `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 $\widetilde{\mathrm{F}}$ language, such as only allowing a limited form of recursion (using `ifold` and `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 $\mathrm{d}\widetilde{\mathrm{F}}$ 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].
- $\widetilde{\mathrm{dF}}$: The code generated by $\widetilde{\mathrm{dF}}$ 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 $\widetilde{\mathrm{dF}}$ (denoted by $\widetilde{\mathrm{dF}}$ (Opt)). The performance of these programs is improved further when the generated C code uses DPS for stack-discipline memory management (denoted by $\widetilde{\mathrm{dF}}$ (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 $\widetilde{\mathrm{dF}}$ ($\widetilde{\mathrm{dF}}$ (Opt)) show similar performance. This shows that the optimisations explained in Section 5 have succesfully made the forward-mode AD code of $\widetilde{\mathrm{dF}}$ as efficient as the reverse-mode AD code. Finally, the performance of $\widetilde{\mathrm{dF}}$ 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 $\widetilde{\mathrm{dF}}$. Both Theano and TensorFlow have performance overhead, which again reduces for bigger data sizes. Specifically, TensorFlow becomes as good as Tapenade (R), Futhark, and $\widetilde{\mathrm{dF}}$.

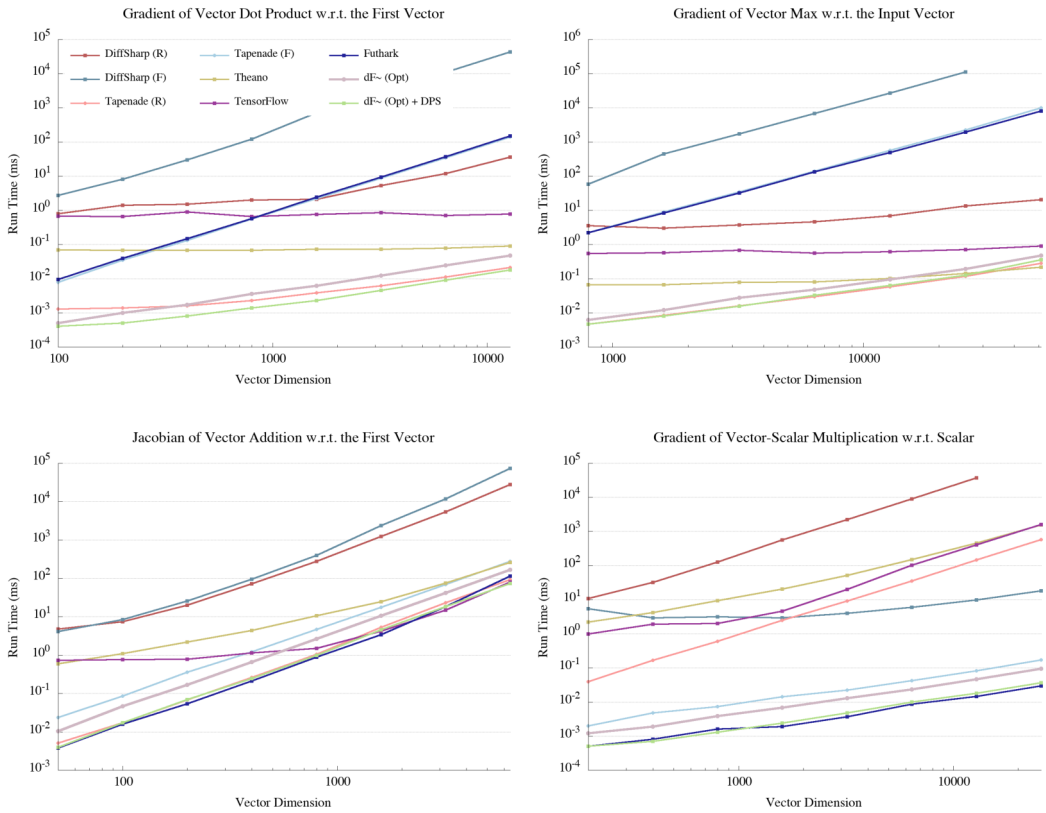

Fig. 9. Performance results for Micro Benchmarks.

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 (NNMF) 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 NNMF 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 NNMF 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.

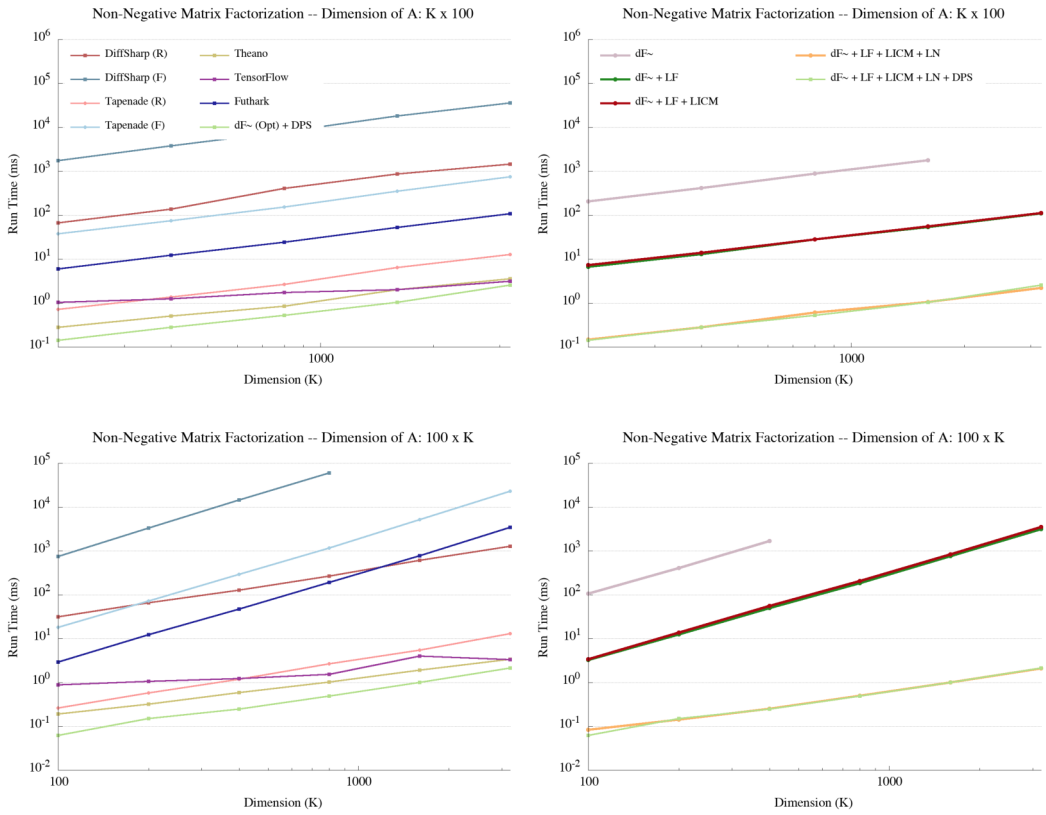

Fig. 10. Performance results for NNMF.

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:

$$\mathcal{D}\left(A||\widetilde{A}\right) = \Sigma_{(i,j)}\left(\log\left(\widetilde{A}_{i,j}\right) + \frac{A_{i,j}}{\widetilde{A}_{i,j}}\right), \qquad \widetilde{A} = WH$$

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. $\widetilde{\text{dF}}$ automatically derives the gradient of the negative log of the likelihood function for the exponential distribution. After performing optimizations, $\widetilde{\text{dF}}$ produces an expression which is equivalent to the following update formula, which is manually derived by hand in [Liu et al. 2010]:

$$\frac{\partial \mathcal{D}}{\partial H} = W^T\left(\frac{1}{WH} - \frac{A}{(WH)^2}\right)$$

Figure 10 shows the performance results of executing the derived update rule on DiffSharp, Tapenade, Theano, and $\widetilde{\text{dF}}$. 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 $\widetilde{\text{dF}}$, we observe that the reverse-mode AD of Tapenade behaves similarly to $\widetilde{\text{dF}}$. This shows that $\widetilde{\text{dF}}$ 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 $\widetilde{\text{dF}}$ 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 $\widetilde{\text{dF}}$, 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, thse transformation rules should be combined with loop fission to become effective. Finally, the DPS representation slightly improves the performance by using the stack-allocation descipline 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:

$$
\begin{aligned}
\text{project}(r, C, f, x_0, k, X) &= \text{distort}(k, \text{p2e}(\text{rodrigues}(r, X - C)))f + x_0 \\
\text{distort}(k, x) &= x(1 + k_1||x||^2 + k_2||x||^4) \\
\text{p2e}(X) &= X_{1..2} \div X_3 \\
\text{rodrigues}(r, X) &= X\cos(\theta) + (v \times X)\sin(\theta) + v(v^T X)(1 - \cos(\theta)), \\
&\quad \theta = ||r||, v = \tfrac{r}{||r||}
\end{aligned}
$$

In order to better demonstrate the expressibility and conciseness of $\widetilde{\text{M}}$, 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. $\widetilde{\text{dF}}$ outperforms all its competitors by up to three orders of magnitude. More specifically, $\widetilde{\text{dF}}$ 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

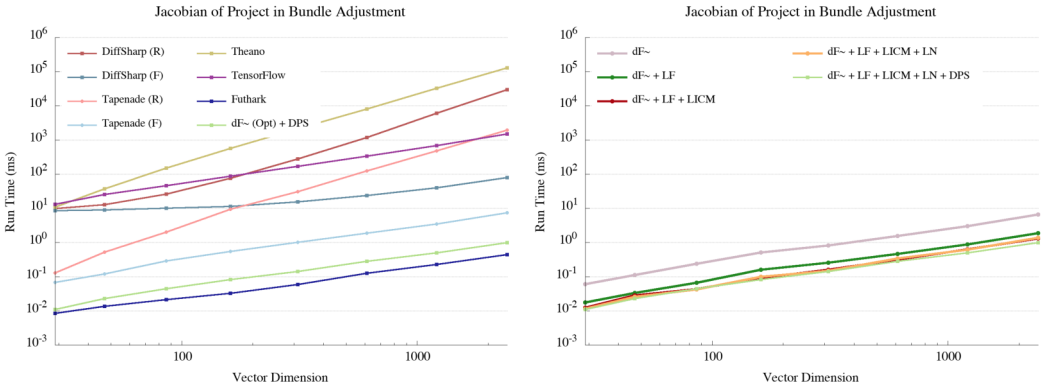

Fig. 11. Performance results for Project in Bundle Adjustment.

```
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 w 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 $\widetilde{\text{M}}$.

C code of $\widetilde{\text{dF}}$ mainly thanks to a better data layout representation for tensors, which we plan to integrate into $\widetilde{\text{dF}}$ 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 $\widetilde{\mathrm{dF}}$, 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, $\widetilde{\mathrm{dF}}$ 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 $\widetilde{\mathrm{dF}}$ solves the perturbation confusion problem using a static approach thanks to the `deriv` macro (Section 4.3). One of the main advantages of $\widetilde{\mathrm{dF}}$ 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, espcially loop transformations.

D* [Guenter 2007] is a symbolic differentiation system, which performs the factorisation of the common product terms in sum-of-product expressions. $\widetilde{\mathrm{dF}}$ also performs a similar idea by performing common-subexpression elimination and loop-invariant code motion in order to eliminate the redundant computation, hence, imporving 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 $\widetilde{\mathrm{dF}}$ 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 acheived 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 probablistic programming, which can be a future direction for $\widetilde{\mathrm{dF}}$.

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 $\widetilde{\mathrm{F}}$. 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 algbra 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 $\widetilde{\mathrm{dF}}$ in the future.

8 CONCLUSIONS

In this paper we have demonstrated how to efficiently compute the derivate 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 $\widetilde{\mathrm{dF}}$ 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

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

Sameer Agarwal, Noah Snavely, Steven M Seitz, and Richard Szeliski. 2010. Bundle adjustment in the large. In *European conference on computer vision*. Springer, 29–42.

Eugene Agichtein, Eric Brill, and Susan Dumais. 2006. Improving Web Search Ranking by Incorporating User Behavior Information. In *SIGIR*.

Johan Anker and Josef Svenningsson. 2013. An EDSL approach to high performance Haskell programming. In *ACM Haskell Symposium*. 1–12.

Ricardo Baeza-Yates, Carlos Hurtado, and Marcelo Mendoza. 2004. Query Recommendation Using Query Logs in Search Engines. In *EDBT*.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015b. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767* (2015).

Atilim Gunes Baydin, Barak A Pearlmutter, and Jeffrey Mark Siskind. 2015a. Diffsharp: Automatic differentiation library. *arXiv preprint arXiv:1511.07727* (2015).

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: A CPU and GPU math compiler in Python. In *Proc. 9th Python in Science Conf.* 1–7.

Michael W. Berry, Murray Browne, Amy N. Langville, V. Paul Pauca, and Robert J. Plemmons. 2006. Algorithms and applications for approximate nonnegative matrix factorization. In *Computational Statistics and Data Analysis*.

Christian Bischof, Peyvand Khademi, Andrew Mauer, and Alan Carle. 1996. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science and Engineering* 3, 3 (1996), 18–32.

Christian H Bischof, HM Bucker, Bruno Lang, Arno Rasch, and Andre Vehreschild. 2002. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. IEEE, 65–72.

Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization *(PLDI '12)*. ACM, 111–120.

Koen Claessen, Mary Sheeran, and Bo Joel Svensson. 2012. Expressive Array Constructs in an Embedded GPU Kernel Programming Language *(DAMP '12)*. ACM, NY, USA, 21–30.

Cliff Click. 1995. Global code motion/global value numbering. *ACM Sigplan Notices* 30, 6 (1995), 246–257.

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

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*. ACM, 151–160.

Frédéric De Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. 2009. Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*. ACM, 729–736.

Zachary DeVito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Nießner. 2016. Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging. *arXiv preprint arXiv:1604.06525* (2016).

Conal Elliott. 2018. The Simple Essence of Automatic Differentiation. *Proc. ACM Program. Lang.* 2, ICFP, Article 70 (July 2018), 70:1–70:29 pages.

Conal M Elliott. 2009. Beautiful differentiation. In *ACM Sigplan Notices*, Vol. 44. ACM, 191–202.

Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *ACM SIGPLAN Notices*, Vol. 28. ACM, 237–247.

Shaun A Forth. 2006. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)* 32, 2 (2006), 195–222.

Jeremy Gibbons. 2006. Fission for Program Comprehension. In *Mathematics of Program Construction*, Tarmo Uustalu (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 162–179.

Andrew Gill, John Launchbury, and Simon L Peyton Jones. 1993. A short cut to deforestation *(FPCA)*. ACM, 223–232.

Clemens Grelck and Sven-Bodo Scholz. 2006. SAC—A Functional Array Language for Efficient Multi-threaded Execution. *Int. Journal of Parallel Programming* 34, 4 (2006), 383–427.

Brian Guenter. 2007. Efficient Symbolic Differentiation for Graphics Applications. *ACM Trans. Graph.* 26, 3, Article 108 (July 2007). https://doi.org/10.1145/1276377.1276512

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.* 39, 3, Article 20 (May 2013), 43 pages.

Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 556–571.

Florent Hivert and N Thiéry. 2004. MuPAD-Combinat, an open-source package for research in algebraic combinatorics. *Sém. Lothar. Combin* 51 (2004), 70.

Robin J. Hogan. 2014. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.* 40, 4, Article 26 (July 2014), 16 pages.

Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4 (Dec. 1996).

Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9, 4 (1999), 355–372.

Kenneth E Iverson. 1962. A Programming Language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*. ACM, 345–351.

Hong-Jiang Zhang Ji-rong Wen, Jian-Yun Nie and. 2002. Query Clustering Using User Logs. *ACM Transactions on Information Systems* 20, 1 (2002).

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.

Manohar Jonnalagedda and Sandro Stucki. 2015. Fold-based Fusion As a Library: A Generative Programming Pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. ACM, 41–50.

Jerzy Karczmarczuk. 1999. Functional differentiation of computer programs. *ACM SIGPLAN Notices* 34, 1 (1999), 195–203.

Kamil A Khan and Paul I Barton. 2015. A vector forward mode of automatic differentiation for generalized derivative evaluation. *Optimization Methods and Software* 30, 6 (2015), 1185–1212.

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 285–299.

Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

Kenneth Levenberg. 1944. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics* 2, 2 (1944), 164–168.

Chao Liu, Hung-chih Yang, Jinliang Fan, Li-Wei He, and Yi-Min Wang. 2010. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*. ACM, 681–690.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. 2015. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*.

Donald W Marquardt. 1963. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics* 11, 2 (1963), 431–441.

Jorge J Moré. 1978. The Levenberg-Marquardt algorithm: implementation and theory. In *Numerical analysis*. Springer, 105–116.

Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. 2010. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science* 1, 1 (2010), 1845–1853.

Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017a. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming:*

Concepts and Experiences (GPCE 2017). ACM, New York, NY, USA, 131–145.

Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2017b. Unifying Analytic and Statically-typed Quasiquotes. *Proc. ACM Program. Lang.* 2, POPL, Article 13 (Dec. 2017), 33 pages.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

Barak A Pearlmutter and Jeffrey Mark Siskind. 2007. Lazy multivariate higher-order forward-mode AD. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 155–160.

Barak A Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 7.

Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2 (2005), 232–275.

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines *(PLDI '13)*.

Jarrett Revels, Miles Lubin, and Theodore Papamarkou. 2016. Forward-mode automatic differentiation in Julia. *arXiv preprint arXiv:1607.07892* (2016).

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *the ninth international conference on Generative programming and component engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136.

Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing Data Structures in High-level Programs: New Directions for Extensible Compilers Based on Staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 497–510.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming* 28 (2018), e10.

Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing Style for Efficient Memory Management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 12–23.

Amir Shaikhha and Lionel Parreaux. 2019. Finally, a Polymorphic Linear Algebra Language. In *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP'19)*.

Jeffrey Mark Siskind and Barak A Pearlmutter. 2005. Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. (2005).

Jeffrey Mark Siskind and Barak A Pearlmutter. 2008. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation* 21, 4 (2008), 361–376.

Daniele G Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *CGO '16*. ACM.

Suvrit Sra and Inderjit S. Dhillon. 2006. *Nonnegative matrix approximation: algorithms and applications.* Technical Report.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-level Functional Expressions to High-performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 205–217.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning *(ICML '11)*. 609–616.

Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions *(ICFP '02)*. ACM, 124–132.

Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays *(FHPC '14)*. ACM, NY, USA, 43–52.

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. 1999. Bundle adjustment—a modern synthesis. In *Inter. workshop on vision algorithms*. Springer, 298–372.

Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP'88*. Springer, 344–358.

Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *Advances in Neural Information Processing Systems*. 10200–10211.

Matthew J Weinstein and Anil V Rao. 2016. Algorithm: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading. *ACM Trans. Math. Softw* (2016).
Christopher Zach. 2014. Robust bundle adjustment revisited. In *European Conference on Computer Vision*. Springer, 772–787.