Author version

Published as:

Sperry, M., Kondap, K., & Hwang, J. T. (2023). Automatic adjoint sensitivity analysis of models for large-scale multidisciplinary design optimization. In AIAA AVIATION 2023 Forum (Paper No. AIAA 2023-3721). https://doi.org/10.2514/6.2023-3721

BibTeX for this reference:

https://lsdolab.github.io/lsdo-publications/publication/sperry2023automatic/

```
@inproceedings{sperry2023automatic,
  author = {Mark Sperry and Kavish Kondap and John T. Hwang},
  title = {Automatic adjoint sensitivity analysis of models for large-scale
    multidisciplinary design optimization},
  booktitle = {AIAA AVIATION 2023 Forum},
  year = {2023},
  doi = {10.2514/6.2023-3721},
  url = {https://doi.org/10.2514/6.2023-3721}
}
```

Automatic adjoint sensitivity analysis of models for large-scale multidisciplinary design optimization

Mark Z. Sperry*
University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093

Kavish Kondap†
Francis Parker School, 6501 Linda Vista Rd, San Diego, CA 92111

John T. Hwang‡
University of California San Diego, 9500 Gilman Dr, La Jolla, CA 92093

Large-scale multidisciplinary design optimization (MDO) is the solution of numerical optimization problems involving dozens or more design variables and multidisciplinary models. It can provide a systematic approach for dealing with high-dimensional design problems where the engineer's intuition is of limited value. One of the critical barriers is the complexity of implementation for adjoint sensitivity analysis which is needed for efficient large-scale MDO. Recently, a new modeling language, called CSDL, was developed that enables the construction of a computational graph for the model. The graph can be used to implement automatic adjoint sensitivity analysis, but the proposed compiler for CSDL incurs significant computational overhead as the cost of the automation. This paper describes a method that eliminates this overhead in the CSDL compiler by using a code generation technique that utilizes graph traversal algorithms. Together, with CSDL, this method bears similarities to reverse-mode algorithmic differentiation but has notable differences such as the application of the implicit function theorem and the use of hybrid dense-sparse linear algebra. The proposed method shows an order of magnitude improvement in computation time compared to the previous compiler. The results indicate that this method is an effective method for sensitivity analysis of large-scale MDO models that achieves both efficiency and automation.

I. Nomenclature

G	=	computational graph
F	=	set of variables to take derivatives of
U	=	set of variables to take derivatives with respect to
$P(x)$	=	predecessors of node x in G
$S(x)$	=	successors of node x in G

II. Introduction

Multidisciplinary design optimization (MDO) is the formulation and solution of numerical optimization problems to aid in engineering design, where the model involves multiple models. The models involved in MDO algorithms are often constructed by assembling a diverse variety of sub-models for different disciplines, subsystems, design conditions, missions, and so on. Hence, the overall multidisciplinary model can be described as heterogeneous.

Large-scale MDO focuses specifically on the subset of MDO problems where there are dozens or more design variables, leading to a high-dimensional design problem. To avoid the curse of dimensionality, gradient-based optimization algorithms need to be used for this class of problems, which makes efficient and accurate sensitivity

*Ph.D. Student, Department of Mechanical and Aerospace Engineering, AIAA student member
†High School Student, AIAA High School Student Member
‡Assistant Professor, Department of Mechanical and Aerospace Engineering, AIAA senior member

analysis an important consideration. The presence of both a large number of design variables and many disciplines, contributing to a heterogeneous model, compound the challenges of efficient and accurate sensitivity analysis.

The complex models involved in MDO problems motivate the use of MDO frameworks to assemble the model and solve the resulting optimization problem. MDO frameworks are software environments for constructing models that facilitate the assembly of larger models from sub-models, interfacing to optimization algorithms, application of common numerical solvers, and visualization, among other tasks.

Frameworks specifically designed for large-scale MDO have an additional benefit that part of the effort to implement sensitivity analysis models can be automated. Two examples of such frameworks are OpenMDAO [1] and GEMSEO [2]. These frameworks rely on the model developer to implement code to compute the partial derivatives of components of their model and then centrally combine these to form total derivatives—the derivatives ultimately needed for gradient-based optimization. However, the computation of partial derivatives is time- and skill-intensive and currently forms the bottleneck for large-scale MDO.

Recently, this bottleneck was overcome by the development of a modeling language called the Computational System Design Language (CSDL) [3]. With CSDL, a three-stage compiler automates the computation of both the partial and total derivatives. However, their approach is based on the modular analysis and unified derivatives (MAUD) architecture which causes it to incur large computational overhead (section III.A) that is overcome by the method presented here [4].

This paper presents a method centered around an algorithm that automatically generates efficient reverse-mode derivative computation code, building on the framework proposed by Gandarillas et al. [3] and decreases computation time by an order of magnitude compared to their MAUD-based approach. Together, with CSDL, this method bears similarities to reverse-mode algorithmic differentiation but has the following notable differences. We leverage the implicit function theorem to efficiently propagate derivatives through nonlinear solvers, something done by only a handful of AD libraries. We also take advantage of known sparsity patterns of mathematical operators to generate code that uses a hybrid of dense and sparse matrix algebra through a novel scheme. Finally, we use the structure of the computational graph to limit unnecessary derivative propagations.

In the next section, we review MDO frameworks, AD methods and implicit differentiation in detail. Then we describe our methodology, and finally present numerical experiments on seven real-life applications that uses our method.

III. Background

Prior to presenting our methodology, we provide a brief overview of modeling frameworks for multidisciplinary optimization (MDO), automatic differentiation methods, and the implicit function theorem as well as its applications.

A. Modeling frameworks for MDO

A modeling framework is a software library, together with a software interface or a modeling language, that facilitates the construction of complex models. Models used in MDO algorithms are often complex because they capture multiple subsystems of disciplines. Further, gradient-based MDO often motivates modular approaches for performing sensitivity analysis that assemble derivatives from sub-models. These properties make it valuable and important to use a modeling framework to construct the model in MDO settings.

One of the widely used MDO modeling frameworks is OpenMDAO, a Python framework that allows users to easily build models of coupled systems in an hierarchical and object-oriented manner [1]. This approach is well-suited and intuitive for MDO problems as they often require considerable coordination between systems. OpenMDAO also automatically computes total derivatives using user-provided partial derivatives for each system [1].

Another well-known software is GEMSEO, a robust MDO design software also written in Python [2]. Like OpenMDAO, GEMSEO allows easily modeling of MDO problems with automatic derivative computations. In addition, it allows users to pick and automatically apply MDO solution architectures [2].

Finally, the three-stage compiler framework introduced by Gandarillas et al. allows for efficient modeling of MDO models and is the framework we base our work on [3]. We will briefly review the architecture surrounding the Computational System Design Language (`CSDL`) upon which our method is constructed.

`CSDL` is a novel algebraic modeling language that leverages a graph-based, three-stage compiler framework to automatically compute sensitivities as depicted in figure 1. The first stage of the compiler consists of the user writing `CSDL` source code that models their MDO problem and an intermediate representation of the given MDO model is built in the form of a directed acyclic graph. Then in the second stage, the graph is transformed to increase efficiency. This graph data structure contains dependency information for all mathematical operations in the model, which is then used

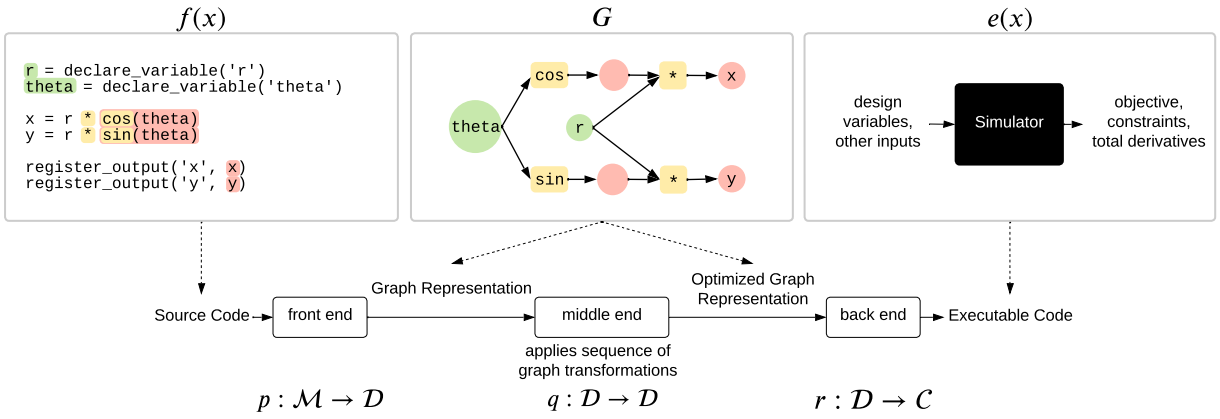

Fig. 1 From Gandarillas et al., the three stage compiler used for CSDL [3].

to build an executable that evaluates the model and its derivatives. This final part, the third stage or what we call the back end, is the area of focus of this paper.

Essentially, the back end performs the important task of evaluating the computational graph and computing its derivatives. In Gandarillas et al., the back end implementation is achieved by building an OpenMDAO problem. Specifically, this is accomplished by defining all mathematical operations as OpenMDAO components along with their partial derivatives and leveraging the total derivative computation offered by OpenMDAO's implementation of the modular analysis and unified derivatives (MAUD) architecture to get access to sensitivities [1].

Thus, unlike pure OpenMDAO, users do not need to provide any partial derivatives which can significantly decrease implementation complexity for the user. This is demonstrated in figure 2 from Gandarillas et al. [3] that shows the number of lines of code needed to implement physics-based wing and rotor aerodynamic analysis tools using CSDL and OpenMDAO.

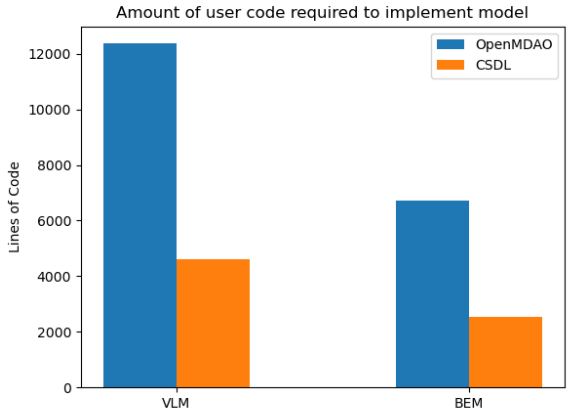

Fig. 2 From Gandarillas et al., the number of lines of code needed to implement a blade element momentum (BEM) and vortex lattice method (VLM) solver in CSDL and OpenMDAO [3].

B. Automatic differentiation

As stated in the introduction, gradient-based optimization requires computing derivatives which is often non-trivial. Specifically, optimization requires the constraint Jacobian and objective gradient with respect to design variables. One simple approach is to use a finite-difference approximation. Although it is straightforward to implement, it is inefficient due to the need to evaluate the model once for each input. Further, it has non-trivial truncation error [5]. Alternatively,

symbolic differentiation involves deriving the exact mathematical expression for derivatives. However, this approach can be very expensive as the expression can become large.

Automatic differentiation (AD) is an alternative approach that involves propagating derivatives through each low-level operation in a program [6]. A significant advantage of the AD technique is that it eliminates the need for user-defined derivatives, while also offering accurate values and relatively fast computational speed [7] [8]. There are two key mathematical formulations for automatic differentiation: forward mode and reverse mode. Forward-mode differentiation consists of applying the chain rule in the same direction of the program evaluation procedure. On the other hand, reverse-mode differentiation consists of computing derivatives from the outputs of the function and successively applying the chain rule backwards. Consequently, the latter method usually requires a full evaluation of the program prior to computing derivatives, unlike forward mode differentiation which does not have such a limitation.

The computational complexity of each mode can be shown by relating them to the computation of the total derivative Jacobian $J \in \mathbb{R}^{m \times n}$ where n denotes the number of inputs and m represents the number of outputs. In the context of optimization, the total derivative Jacobian is the derivatives of the objective and constraints with respect to design variables. Forward mode differentiation can be generalized as computing one column of J per sweep while one reverse mode sweep computes one row of J a time [8]. Thus, reverse mode is generally more efficient than forward mode when $m < n$ and vice versa. Finally, we draw attention to optimal Jacobian accumulation, a hybrid approach using both forward mode differentiation and reverse mode differentiation that can be more efficient than both methods [9]. Unfortunately, optimally performing this procedure has been proven to be NP-complete although there are approximation algorithms that attempt to solve it [9].

There are two ways to implement AD: source code transformation and operator overloading [5]. Source code transformation generates source code of the derivative computation used for AD. On the other hand, operator overloading, as the name suggests, overloads math operators to perform derivative computation in addition to their original behavior [7]. AD frameworks can also be classified as static or dynamic [10]. Static frameworks build a computational graph once and is used repeatedly. This is beneficial as expensive graph optimizations can be performed once during the compile stage. In contrast, dynamic frameworks regularly rebuild the computational graph during runtime [10]. Although more expensive, dynamic frameworks are often easier to debug and offer more flexible control flow.

Automatic differentiation is a well-established domain of research that has been studied extensively for decades and has been used for statistical analysis [7] and gradient-based optimization. In recent years, the popularity of AD has increased significantly due to its relevance in the domain of machine learning. One of the most popular implementations is TensorFlow, an open-source Google library made for deep learning that creates a static optimized computational graph with AD capabilities [11]. Chainer, another framework for machine learning, also uses a static graph structure and uses back-propagation (reverse mode differentiation) for AD [12]. On the other hand, PyTorch is a dynamic framework that implements reverse mode differentiation using operator overloading [10]. Similarly, TensorFlow Eager is a feature of TensorFlow [13] that also allows dynamic construction of the computational graph. Finally, Caffe is another deep learning framework that utilizes back-propagation[14]. The aforementioned libraries are heavily focused on performance and all offer GPU support.

Another high performance library is JAX. A Python library based on Harvard's Autograd [15], JAX's Numpy-like API allows members of the scientific computing community access to efficient derivative computation through JIT compilation [16]. It performs automatic forward and reverse mode differentiation to compute derivatives of any order. Similarly, the Stan Math Library is a robust, general purpose C++ library that can perform forward mode, reverse mode as well as mixed mode differentiation using operator overloading [17]. Aesara (previously known as Theano) is another general purpose tensor-based Python library that allows graph compilation as well as symbolic differentiation [18]. Two powerful libraries not implemented in Python or C++ are the ForwardDiff.jl and ReverseDiff.jl libraries which perform forward mode and reverse mode differentiation respectively in the Julia language [19] [20].

Although all aforementioned libraries are robust and efficient, they are not built with MDO applications in mind. For example, they do not provide a user-friendly interface for implementing subsystems and their connections, as OpenMDAO or GEMSEO do. Further, many of them do not efficiently propagate derivatives through implicit functions. To demonstrate the difference in complexity of MDO models and machine learning models, figure 3 compares a selection of real-world machine learning networks and MDO models that were all implemented in `CSDL`. The machine learning networks are typically written in one of the aforementioned AD libraries. Notably, MDO models use more implicitly defined functions, a larger number of models (subsystems) and deeper model hierarchies.

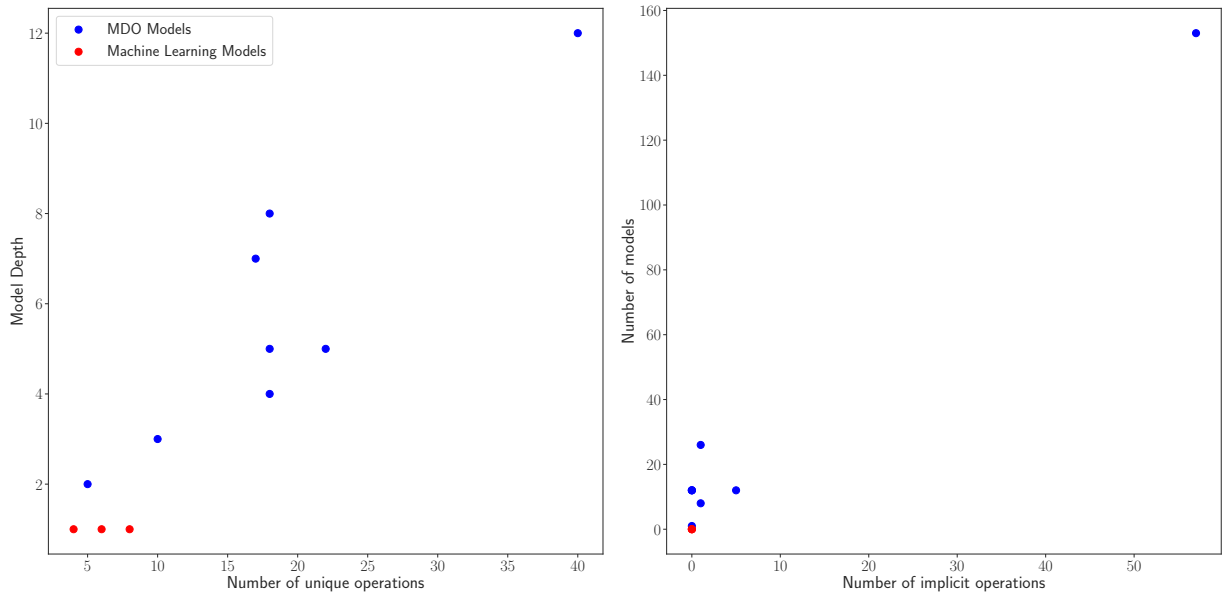

Fig. 3 Differences in complexity between MDO models and machine learning models. The machine learning models (including neural networks, support vector machines, linear regression, and logistic regression) were written and utilized CSDL's automatic differentiation to compute gradient functions.

C. Implicit Function Theorem

A special, and not often considered, case in automatic differentiation is propagating derivatives through implicitly defined mathematical operators. For example, a user may wish to differentiate a program that involves the solution of $F(x, y) = 0$ for y with the goal of computing dy/dx where x has been computed before F is called. Although not as common in machine learning applications, solving an implicitly defined function is common in scientific computing and multidisciplinary design optimization where strong coupling often exists between disciplines. Such functions are often solved using an iterative nonlinear solver such as Newton's method or nonlinear block Gauss-Seidel [4]. However, iterative solvers pose challenges when applying automatic differentiation as derivatives must be propagated through each iteration which leads to large computation costs. Christianson [21] proposed a method to reverse through the graph but to adjust the number of iterations during the reverse sweep. Further, Azmy [22] presented a procedure by differentiating a single additional iteration.

However, an approach that exploits the implicit function theorem has been shown to be an effective use for automatic differentiation. The statement of the implicit function theorem is as follows. Let $F(x, y) : \mathbb{R}^{p+q} \rightarrow \mathbb{R}^q$ be a continuously differentiable function with $\partial F/\partial y$ invertible where $y \in \mathbb{R}^q$ contains the dependent variables and $x \in \mathbb{R}^p$ contains the independent variables. Consider a point (x^*, y^*) where $F(x^*, y^*) = 0$. Let $F_x(x^*, y^*)$ and $F_y(x^*, y^*)$ be

$$F_x(x^*, y^*) = \begin{bmatrix} \frac{\partial F_0}{\partial x_0}(x^*, y^*) & \dots & \frac{\partial F_0}{\partial x_p}(x^*, y^*) \\ \vdots & & \vdots \\ \frac{\partial F_q}{\partial x_0}(x^*, y^*) & \dots & \frac{\partial F_q}{\partial x_p}(x^*, y^*) \end{bmatrix}, \qquad F_y(x^*, y^*) = \begin{bmatrix} \frac{\partial F_0}{\partial y_0}(x^*, y^*) & \dots & \frac{\partial F_0}{\partial y_q}(x^*, y^*) \\ \vdots & & \vdots \\ \frac{\partial F_q}{\partial y_0}(x^*, y^*) & \dots & \frac{\partial F_q}{\partial y_q}(x^*, y^*) \end{bmatrix}$$

A result of the implicit function theorem is that

$$\frac{\partial y}{\partial x}(x^*, y^*) = -F_y(x^*, y^*)^{-1} F_x(x^*, y^*). \tag{1}$$

Let x and y denote intermediate variables in a program to which AD is applied. During the forward or reverse accumulation procedure, the partial Jacobian $\partial y/\partial x$ of an operation can be replaced with equation (1). This mandates a linear system to be solved instead of explictly computing $\partial y/\partial x$ which can save significant computational costs. Notably, this approach has been implemented in software packages such as JAXopt, which builds on JAX [23], and

ImplicitAD.jl*, which builds on Julia's AD framework.

IV. Methodology

In this paper, we present a novel method for the back end in three-stage compiler proposed by Gandarillas et al., which was described in section III.A. The back end exploits the static graph structure to build an executable that efficiently evaluates the model and computes its derivatives using reverse mode differentiation as detailed in section III.B. Finally, we use the implicit function theorem to propagate derivatives through operations using nonlinear solvers using the implicit function theorem which was presented in section III.C.

A. Computational Graph

The computational graph G is a static representation of the user-defined Computational System Design Language `CSDL` `Model` and is only built and processed once during the compilation step, allowing graph optimizations not normally available with dynamic graph frameworks. It is in this compilation step that executables for model evaluation and derivative computation are built and is the subject of our main contribution. Henceforth, we assume that G has already been built according to Gandarillas et al. [3]

We proceed by defining the structure of the computational graph G as implemented in `CSDL`. The structure is similar to that of Aesera (Theano) [18], where G is directed, acyclic and bipartite. Nodes represent either multi-dimensional arrays or mathematical operators on them denoted as `Variables` and `Operations` respectively. Directed edges represent dependencies between `Operations` and its inputs and outputs which are `Variables`. An `Operation` node always has at least one `Variable` node in its set of predecessors (inputs to the operation) and successors (outputs to the operation), meaning leaf nodes of G are always `Variables`. Further, `Variable` nodes may have multiple `Operation` successors but only zero or one `Operation` predecessor. An example can be seen in figure 1.

As mentioned in the background section, an `Operation` node can generally be one of three types: `StandardOperation`, `CustomOperation`, or `ImplicitOperation`. A `StandardOperation` is the most basic type and consists of explicitly defined mathematical functions fully defined in `CSDL` such as addition, multiplication, trigonometric operations, or matrix-multiplication and so on. For these operations, the partial derivative Jacobian is automatically determined. For external codes and special operations, a `CustomOperation` can be defined by the user. The user provides a Python method for evaluating the operation and a method that computes its partial derivatives given its input values. They have the freedom to utilize tools beyond Numpy, as long as derivative information is provided. Finally, an `ImplicitOperation` is an implicit function defined by a residual as defined in equation (1). These nodes contain their own computational graph instances defining the calculation of the residual as defined in Gandarillas et al. [3] as well as their own non-linear solver.

B. Model Evaluation

Calling the executable of the model evaluation performs all computations of the user-defined `CSDL` `Model` given a set of input values. A straightforward code generation technique is used to build a sequential program in an iterative manner. A topological ordering T of the computational graph is determined using an implementation of Kahn's algorithm to guarantee correct outputs. For every `Operation` node in T, the instructions to evaluate the specific mathematical expression is appended to a comprehensive instructions object in order. The appended instruction at each operation depends on the three types of `Operation` nodes as discussed in the previous section. For a `StandardOperation`, the instructions are simple strings of the mathematical operation given the names of the outputs and inputs. A compiled language can be used for the instructions as information on `Variable` shapes are static and known ahead of time. For the implementation used in this paper, Numpy code is used. For a `CustomOperation` node, the instruction involves calling the user-defined Python method while providing the values of its input. Finally, if a node is an `ImplicitOperation`, its specified non-linear solver is invoked as well as its residual evaluation executable from its own computational graph.

Unlike `StandardOperation` nodes which can be defined using pure strings, `ImplicitOperation` nodes have solver objects associated with them. Available non-linear solvers are inspired by those used in OpenMDAO [1]. These include Newton's method, non-linear block Gauss-Seidel and the bracketed search method. Each of these solvers create their own isolated executables of the computational graph defining the residual evaluation as well as its derivatives. The residual evaluation is called as needed by their respective solvers.

*https://github.com/byuflowlab/ImplicitAD.jl

Finally, for any operation that requires it, any constant variables that are defined at compile time and needed for evaluation are provided with the instructions for each operation.

C. Derivative Evaluation

The executable for derivative evaluation provides derivatives at the current state of the model. Unlike model evaluation, the user supplies the names of any variables to take derivatives of as well as any variables to take derivatives with respect to. The provided variables are not limited to leaf nodes and can be any subset of variables in the computational graph. Let the set of variables to take derivatives of be F and the set of variables to take derivatives with respect to be U. Therefore, derivative evaluation computes df_i/du_j for all $f_i \in F$ and $u_j \in U$. As stated previously, it is often the case that F consists of the objective and constraints while U consists of the design variables for multidisciplinary optimization problems. The underlying structure for building the executable is similar to that of model evaluation. Smaller instructions are iteratively added to a comprehensive set of instructions to build a sequential program which can then be compiled.

We now present our general approach for computing the aforementioned total derivatives from the computational graph G. Without loss of generality, we consider the problem of computing df/du for all $u \in U$, where F consists of only a single variable f. A high-level overview of the procedure is shown in algorithm 1. As discussed previously, we generate reverse-mode differentiation code. Before the main graph traversal loop, we initialize several data structures during pre-processing. These include an empty set data structure denoted as `ComputedPartials` which will contain strings of operation names and another empty set that will contain node name strings `VisitedVars`. Further, a static set `UDescendants` which is the union of descendants of u for all $u \in U$ and the ancestors of f called `FAncestors` are computed in linear time with the number of nodes in the graph. Finally, a queue of variable names `Q` initialized as $[f]$ is used for the graph traversal.

Algorithm 1 Generate reverse-mode differentiation code

1: **while** Q **do**
2: Verify whether the successors of the current operation o_v have been processed.
3: If necessary, compute dense/sparse partial Jacobian of o_v depending on operation sparsity structure. (section IV.C.1).
4: Accumulate derivatives through o_v using matrix-multiplication, vjp or implicit propagation (section IV.C.2).
5: Update Q with newly available nodes based on dependency relationship of o_v and U (section IV.C.3).
6: **end while**

The main loop is a standard breadth-first traversal of G from f with some minor modifications. Firstly, we traverse G in the opposite direction of the directed edges. Additionally, we use `VisitedVars` to traverse the graph in reverse topological order. The current variable being processed will be the front, removed element of Q and is denoted as v while the predecessor operation that computes v is denoted as o_v.

At each iteration of the traversal, there are three main steps: compute partial derivatives, accumulate the Jacobians, and update available nodes. We provide an overview of the fundamental steps and present a more thorough explanation in subsequent sections. We begin by immediately adding v to `VisitedVars` and checking if all successors of o_v exists in `VisitedVars` or not present in `FAncestors`. If the preceeding checks do not pass, the outputs of o_v have not been fully visited for reverse accumulation, triggering a move to the next iteration of the graph traversal. Otherwise, we continue with accumulation by computing the partial derivative Jacobians of o_v (if required). Then, we perform the matrix-multiplication or accumulation procedure governed by the chain rule to propagate derivatives through to the inputs of o_v. Finally, we update Q with any newly available variables.

1. Partial derivatives

For `StandardOperation` nodes, explicit partial derivative matrices are used for the derivative accumulation as per the chain rule. Similar to model evaluation, the instructions are strings of code that evaluate the mathematical expression of the operation's partial derivative matrix. The instructions are added to the comprehensive derivative instruction set as needed. To avoid unnecessary recomputations, the operation is added to `ComputedPartials` whenever the partials are added to the instruction set. When an operation's partials are needed for accumulation again, a check to see if the operation is already in `ComputedPartials` is done.

A large percentage of the `StandardOperation` set are elementwise functions which are characterized by a direct

index mapping between the input and output tensors. Examples include addition, multiplication and trigonometric functions. The partials of such operations have a specialized structure as they are purely diagonal. Repeated construction of the full matrix is expensive; thus we instead build a 1D vector consisting of only the diagonal entries.

There is also the issue in the non-uniformity of tensor sizes of operation input and output variables. Using dense linear algebra for derivative propagation on operators is computationally infeasible for large variables. Conversely, using sparse algebra for small and dense Jacobian products leads to wasteful computations. Fortunately, our static graph framework gives us information on the sparsity structure of `StandardOperation` nodes as well as variable tensor sizes. We leverage this information by determining a sparse or dense representation of Jacobians on an operation-by-operation basis in the compilation stage. Therefore, instructions produced for o_v's derivatives reflect the chose data structure. We note that any value that can be pre-computed is also computed ahead of time. For example, if o_v has a static sparsity structure, the partials matrices are pre-allocated and only the non-zero indices are updated at runtime.

Finally, we note that the aforementioned technique may not be applicable when o_v is a `StandardOperation` instance. For `CustomOperation` nodes, users have an option of providing methods that compute partial derivative matrices. In this case, sparse or dense matrix-multiplication is performed depending on the data structure provided by the user. However, they also have the option of providing a method that performs a vector-Jacobian product. This is equivalent to providing a function that continues the derivative propagation given vector-Jacobian products instead of performing matrix multiplication explicitly. These accumulation functions are also applicable when o_v is an `ImplicitOperation` instance and used for the implicit-function theorem as described in the following section.

2. Reverse accumulation

With partial derivatives of o_v taken into account in the instruction set, the chain rule must be applied. Specifically, the propagated derivatives of f with respect to all outputs (successors) of o_v. This step is done using reverse mode differentiation.

We first give an overview of the approach taken when o_v has an explicit partial derivative matrix defined as per the case described in the preceding section. For each successor variable s in the set $S(o_v)$ of successors to `Operation` node o_v, there is a path accumulation variable $\bar{s}$ associated. The value of $\bar{s}$ represents the derivative df/ds. To propagate derivatives to a predecessor p in the set $P(o_v)$ of predecessors of o_v, we use

$$\bar{p} = \frac{df}{dp} = \sum_{s \in S(o_v)} \bar{s} \frac{\partial s}{\partial p} \tag{2}$$

Instructions describing these matrix multiplications and additions are added to the comprehensive instructions at this stage. The computation for the partial derivatives $\partial s/\partial p$ are already taken into account at this point either from the preceding section or from earlier due to the topologically ordered traversal using `VisitedVars`. Similarly, $\bar{s}$ have been computed for all $s \in S(o_v)$for the same reason as they are simply the left-hand-side of equation 2 for any directly succeeding operations to o_v that are ancestors of f which have already been traversed.

As mentioned previously, any matrix partials of o_v are assumed to be either sparse or dense. The matrix multiplication and addition procedures take any potential differences in data structure into account. Further, elementwise operations have their partials stored as a one-dimensional vector representing their diagonal elements. For such operations, a row-wise multiplication is performed instead of the standard matrix-multiplication, saving costs of unnecessary matrix-multiplications and matrix construction. Finally, if o_v is that of a `CustomOperation`, a method for a vector-Jacobian product may be provided instead of the full partial derivative matrix. For such cases, we use

$$\bar{P}_{i,:} = \text{vjp}(\bar{S}_{i,:}, P), \tag{3}$$

where

$$\bar{P} = \begin{bmatrix} \bar{p}_0 & \dots & \bar{p}_m \end{bmatrix}, \qquad \bar{S} = \begin{bmatrix} \bar{s}_0 & \dots & \bar{s}_n \end{bmatrix},$$

and vjp is the vector-Jacobian product method provided by the user. It is evident that the number of vjp calls to build $\bar{P}$ is equal to the number of rows in $\bar{S}$ which is equivalent to the size of f. To obtain the accumulated derivatives of each variable $\bar{p}$, we extract the relevant columns from $\bar{P}$.

More care for computational efficiency is taken for functions defined implicitly using `ImplicitOperation` instances. Here, let R denote the function computing the residual in o_v. The implicit function theorem is applied as per equation 1:

$$\bar{P} = -\bar{S} R_S(S^*, P^*)^{-1} R_P(S^*, P^*). \tag{4}$$

We then can obtain a linear system:

$$\bar{P}^T = -R_P(S^*, P^*)^T \left(R_S(S^*, P^*)^{-T} \bar{S}^T \right), \tag{5}$$

where

$$R_S(S^*, P^*) = \begin{bmatrix} \frac{\partial R_0}{\partial s_0}(S^*, P^*) & \dots & \frac{\partial R_0}{\partial s_n}(S^*, P^*) \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial s_0}(S^*, P^*) & \dots & \frac{\partial R_n}{\partial s_n}(S^*, P^*) \end{bmatrix}, \qquad R_P(S^*, P^*) = \begin{bmatrix} \frac{\partial R_0}{\partial p_0}(S^*, P^*) & \dots & \frac{\partial R_0}{\partial p_m}(S^*, P^*) \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n}{\partial p_0}(S^*, P^*) & \dots & \frac{\partial R_n}{\partial p_m}(S^*, P^*) \end{bmatrix},$$

and S^* and P^* denote values at the fixed point $R(S^*, P^*) = 0$. Recall that a separate computational graph exists for R, implying access to executables for the evaluation of R and its derivatives. These are used to compute $R_S(S^*, P^*)$ and $R_P(S^*, P^*)$. The criteria for improved performance of equation 5 over is clear. When solving equation 5, the number of linear systems to solve is equal to the size of f while for equation IV.C.2, it is equal to the total size of P. For MDO problems, it is often the case that f is an objective variable and thus a scalar, meaning equation 5 reduces the number the of linear systems to solve. However, there is certainly a benefit in using equation IV.C.2 in some situations as the linear systems can be solved once independent of f. A linear solver can be specified to solve the system depending on the structure of R. Finally, to get accumulated derivatives of each variable $\bar{p}$, we extract the appropriate sections of the matrix $\bar{P}$.

3. Updating available nodes

At this point, o_v's predecessors have had its derivatives propagated and we update Q to update newly available nodes for the graph traversal. However, it is not necessarily the case that we add all predecessors $p \in P(o_v)$ to Q as p may not necessarily depend on U. For this reason, we only add p to Q if they are in the set `UDescendants`. This behavior combined with the reverse graph traversal ensures we only accumulate operations in the set

$$V_f = \text{Ancestors}(f) \bigcap \left(\bigcup_{u \in U} \text{Descendants}(u) \right), \tag{6}$$

to compute the derivatives $df/du \forall u \in U$.

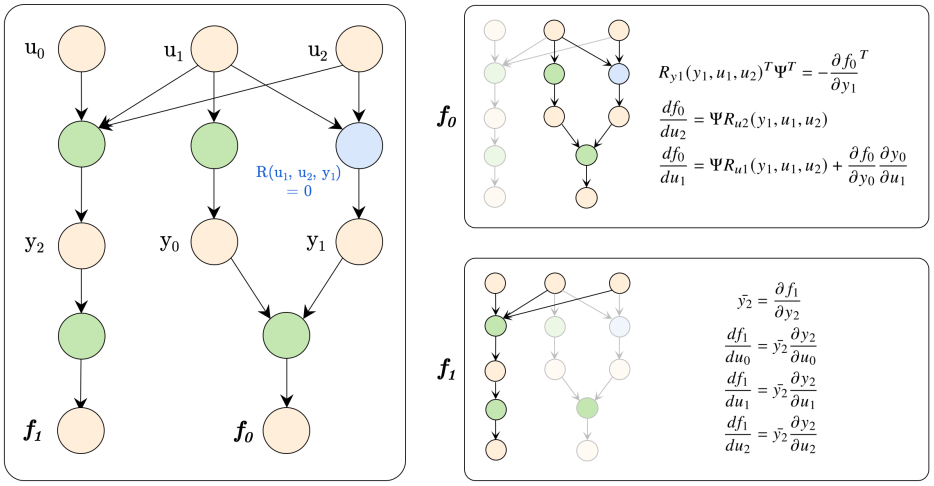

Fig. 4 A simple example showing a computational graph and its generated expression. This model contains one implicitly defined function R as well as two outputs $F = \{f_0, f_1\}$ and three inputs $U = \{u_0, u_1, u_2\}$.

So far, we have made the assumption that F contains only a single variable f. However, it is common to have other variables such as constraints that requires derivatives. For the general case of an arbitrary size F, the above procedures are repeated for all $f \in F$ with the exception of reinitializing `ComputedPartials` and recomputing `UDescendants`.

	model depth	# of nodes	# of unique operations	max variable size	# of models	# of outputs	# of inputs	# of implicit operations	# of linear solves	ratio saved from accumulation	sparse Jacobian ratio
Power-Beaming	2	303	5	1	12	11	8	0	0	0.7116	0.0
NASA Lift Plus Cruise	12	61892	40	1.09E+7	153	19.0	111	57	86	0.9443	0.2194
Blade Element Momentum	4	811	18	180	8	3	15	1	3	0.6206	0.6906
Trajectory Optimization (Time-marching)	5	5043	18	40	12	93	121	0	0	0.9983	0.0
Trajectory Optimization (Collocation)	5	5254	22	156	12	1068	1096	0	0	0.9947	0.0965
Motor Optimization	7	1703	17	25	12	1	2	5	5	0.1755	0.0
Vortex Lattice Method	8	1561	18	1800	26	1	99	1	1	0.2152	0.8669
MNIST dataset	1	25	4	2.51E+7	0	20	7960	0	0	0.3333	0.7778
PDE Optimal Control	3	40101	10	12100	1	1	41	0	0	0.0	0.0001
Logistic Regression	1	33	6	14105	0	1	31	0	0	0.2	0.8333
Support Vector Machine	1	39	8	4.00E+6	0	1	3	0	0	0.125	0.3571

Table 1 Overview of models implemented in CSDL and their problem details. Italicized models indicate machine learning models.

We note that traversing the graph multiple times can potentially lead to redundant operation accumulation procedures if there are variables in F with many ancestors. However, we attempt to reduce this cost by minimizing the number of operations considering only one variable at time. Further, if an operation has explicit partial derivatives, recomputations of partials are avoided using `ComputedPartials`.

Figure 4 shows a trivial example of a computational graph containing an `ImplicitOperation` with three inputs $U = \{u_0, u_1, u_2\}$ and two outputs $F = \{f_0, f_1\}$. The generated code for each output and its derivatives are shown on the right. We demonstrate that the computation of both df_0/du_2 and df_0/du_1 can be achieved with only a single linear system solved. Further, we can see the effects of our graph-traversal algorithm ignoring operations not involved in derivative computation.

V. Numerical Experiments

In this section, we present numerical results from several practical multidisciplinary design optimization problems that showcase the performance gain from our implementation of the methods described in section IV. The collection of problems are real-world examples that span a wide range of classes that have all been successfully integrated into CSDL and solved for various research projects. One problem is the large-scale multidisciplinary optimization (MDO) of NASA's lift-plus-cruise aircraft as studied by Sarojini et al. [24]. Another is the large-scale MDO of an aircraft powered by a laser beam, which is a simplified version of the problem proposed by Orndorff et al. [25]. We also include three discipline analysis tools: a rotor design tool by Ruh and Hwang [26] that uses blade element momentum theory, an aerodynamic solver based on the vortex lattice method, and a low-fidelity motor analysis tool based on an equivalent circuit model. Finally, we include two optimization problems that incorporate time-dependent simulations: a trajectory optimization study of NASA's lift-plus-cruise aircraft by Orndorff et al. [27] and a time-dependent optimization of a non-linear reaction-diffusion reaction. The dynamic simulations use a CSDL-compatible ordinary differential solver that provide adjoint-based derivatives. Implementation details and performance information of each application are displayed in table 1.

In the following sections, we present performance results compared to the implementation by Gandarillas et al. [3] which performs derivative computation based on the MAUD framework [1]. Then, we analyze some outcomes from our method explained in section IV.

1. Performance comparison

In this section, we compare the computational efficiency for model evaluation and derivative computation of our proposed approach and the implementation presented by Gandarillas et al. [3], where we refer to the latter as operation-level MAUD. MAUD, which stands for modular analysis and unified derivatives, is an architecture that computes derivatives for coupled models by solving the unified derivatives equation (UDE) [4] [5]. The implementation of MAUD

has been realized in OpenMDAO [1] as discussed previously. We focus on operation-level MAUD, a special case where where every low-level operation is considered as a separate OpenMDAO component. This contrasts with a traditional use of MAUD, where the model is implemented directly in OpenMDAO, almost always with more manual effort required to compute derivatives and with fewer, larger components, rather than more, smaller components. Operation-level MAUD translates to an implementation where every `Operation` node is modeled as an component in OpenMDAO. This results in an OpenMDAO problem consisting of a combination of `StandardOperation` and `CustomOperation` explicit components and a few `ImplicitOperation` implicit components containing their own OpenMDAO problem instances. Because the entire model is implemented in OpenMDAO, the derivatives are automatically computed using OpenMDAO's implementation of the MAUD architecture by solving the UDE. We solve the seven model problems

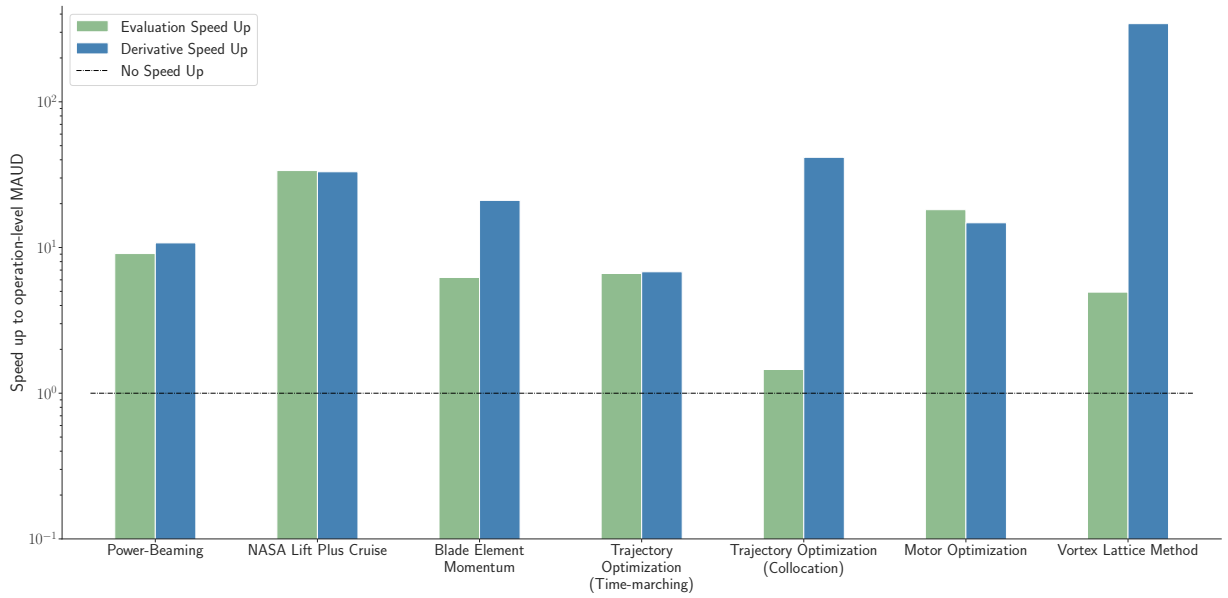

Fig. 5 Performance improvements from our proposed method compared to operation-level MAUD as implemented by Gandarillas et al. on real-life applications. The speed up is computed by dividing the execution time of operation-level MAUD by our approach.

described above in using this architecture and compare it to the method presented in this paper in figure 5.

We observed that there is about an order of magnitude speed up for derivative computation and slightly less for model evaluation. The speed-ups demonstrate that the object-oriented structure of MAUD is a significant bottleneck for computational performance. OpenMDAO creates separate object instances for each component and group. In Python, there are significant overhead costs associated with such a large hierarchy tree of objects which can impede performance of such structures. Such costs can be attributed to the dynamic nature of Python which requires run-time type checking as well as interpreter overhead. Instead of computation through object-oriented programming, our code generation technique builds an almost purely mathematical sequential program that can be compiled before execution. Further, our graph-traversals are done once during compile-time as opposed to dynamic frameworks such as OpenMDAO which does it at run-time.

Although model evaluation of operation-level MAUD and our proposed method differs only in program structure, the two frameworks use fundamentally different approaches for derivative calculation. OpenMDAO solves the UDE to compute total derivatives while our method uses reverse-mode differentiation. Although the problems span a wide range of classes, it can be observed that all problems received a performance improvement by around an order of magnitude and up to two orders of magnitude with the methods presented in section IV. These include cases like the power-beaming example which has more outputs than inputs where reverse-mode differentiation is less effective. The following sections explore these improvements in more depth.

We note that our comparisons are with operation-level MAUD, which was the method discussed and implemented by Gandarillas et al. and is a specific and rather extreme case of MAUD. These comparisons are not with MAUD and OpenMDAO in general. Further work must be done to comprehensively characterize the cause of performance

differences between our approach and MAUD.

2. Impact of hybrid dense-sparse derivative propagation

In section IV.C.1, we discussed automatic sparse partial derivative detection when generating reverse-mode differentiation code. Figure 6 shows improvements of our sparse-dense hybrid approach compared to the case when all partials are dense or sparse for a vortex lattice method aerodynamic solver. On the left, we show the ratio of all

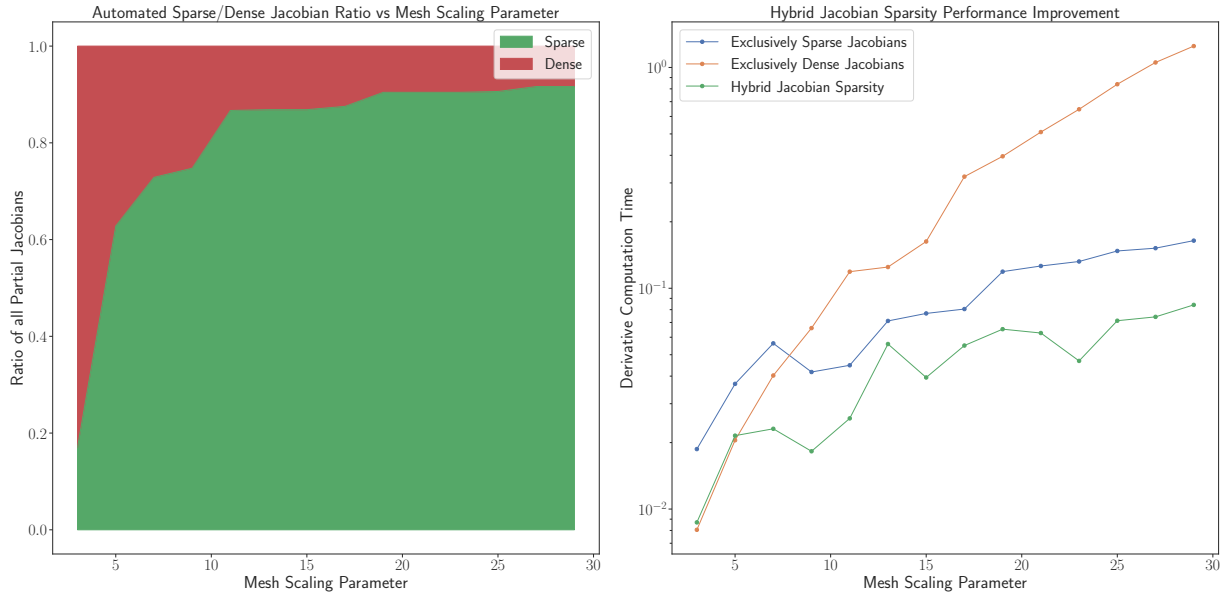

Fig. 6 Performance of derivative computation procedure when operation partial derivative matrices are exclusively sparse or dense compared to our automatic hybrid approach. The hybrid approach automatically makes partial derivatives more sparse as mesh size increases.

operation partial derivatives that are sparse and dense as we increase mesh size. It can be observed that at smaller mesh sizes, a significant amount of partial derivatives are dense due to smaller variables in the model. As we increase the mesh size, variables get larger and in turn, more partials are determined to be sparse. On the right, we observe that our automated hybrid sparse-dense approach has higher efficiency than when partials are exclusively sparse or dense. At low mesh sizes, out hybrid approach matches the optimal dense approach as dense matrices are more computationally efficient compared to their sparse counterparts when matrix dimensions are sufficiently small. However, the exclusively dense formulation scales poorly with mesh size unlike the exclusively sparse formulation. This is due to the partial matrices increasing exponentially with size which is unfavorable for the dense formulation. However, because the model contains some operations independent of mesh size, it can be observed that some operations are better off using dense partials. Our automated hybrid approach takes this into account, leading to improved efficiency.

Figure 7 shows the sparse/dense partial derivative percentage for all seven application problems. We note that there is a heterogeneous distribution of sparsity where some models have a high fraction of dense operations and vice versa. This indicates that our automatic hybrid dense/sparse approach is effective as it can accommodate models with varying degree of variable sizes.

3. Effect of disconnected operations during accumulation

Finally, we investigate the effects of our queue updating mechanism discussed in section IV.C.3 that governs the selection of the next nodes to be processed in our graph traversal. As discussed, our reverse-mode computations only accumulates necessary operations for each f in F. In the MAUD architecture, this approach is not taken and the entire model consisting of all components are processed for each f in F. For operation-level MAUD, this translates to every operation being accumulated n_F times where n_F is the total number of scalars in F. On the other hand, our method guarantees that this does not occur unless every input operation (an operation that has predecessors with in degree

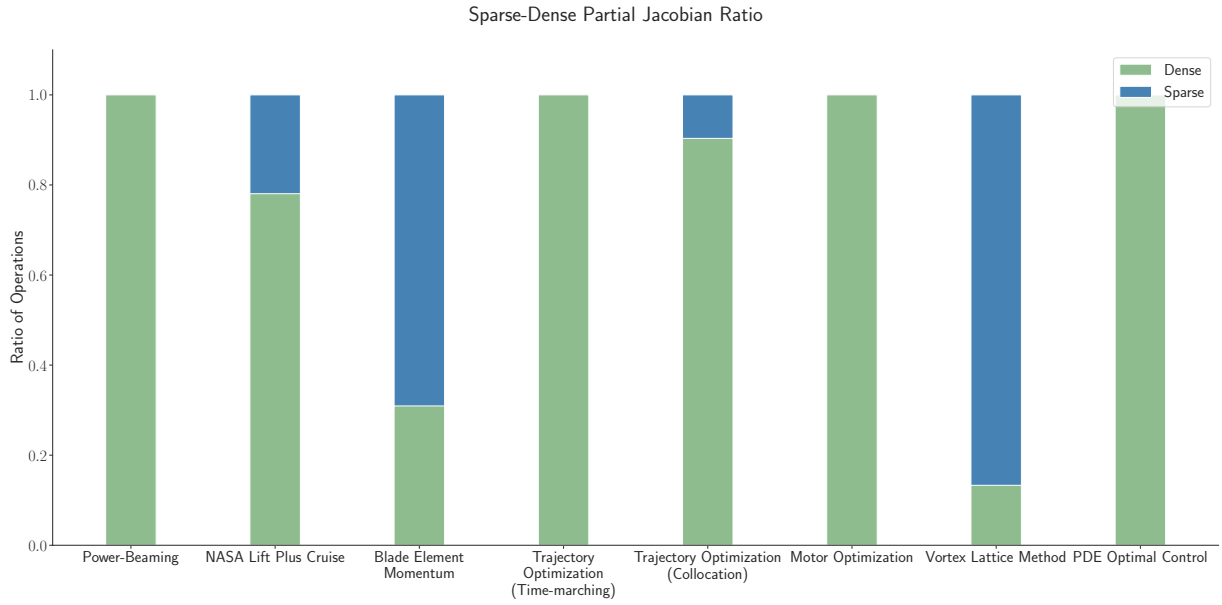

Fig. 7 Ratio of operation partial derivatives that are sparse as opposed to dense for differentiation. Each operation has different factors such as variable input/output size and sparsity patterns that determines if its sparse.

zero) has a predecessor in U and every f in F has every operation in its ancestors. Figure 8 shows the percentage of

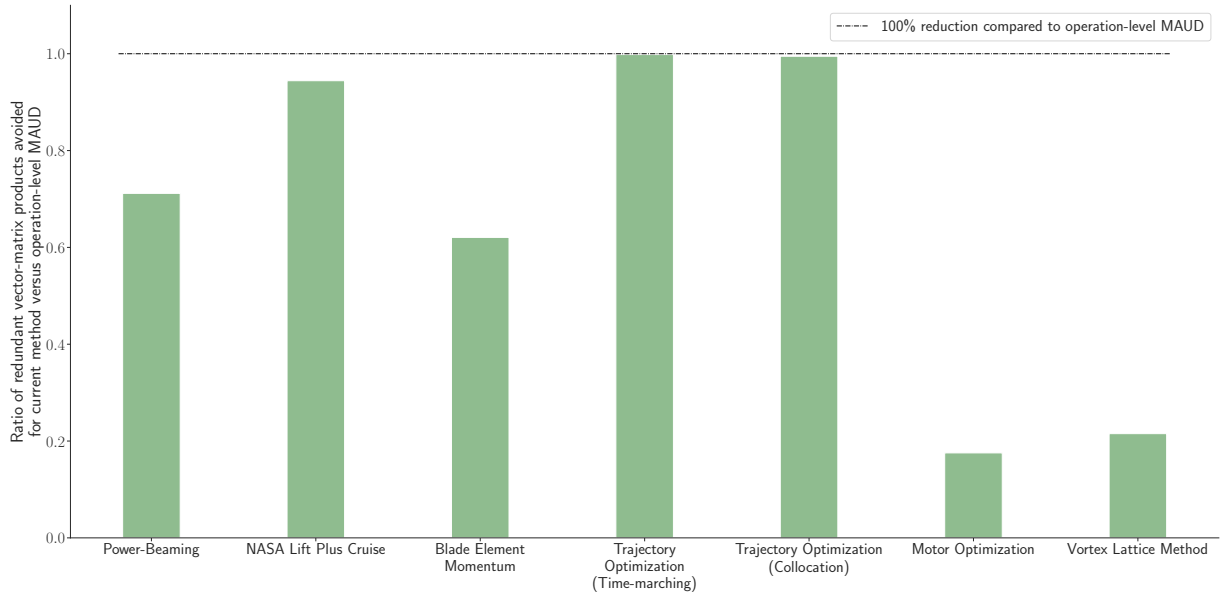

Fig. 8 Ratio of vector-matrix accumulations as defined in equation (2) saved compared to operation-level MAUD.

vector-matrix accumulations saved compared to operation-level MAUD for our seven applications. The percent of vector-matrix accumulations saved is computed from

$$\% \text{ Saved} = \sum_{f \in F} \left(1 - \frac{n_{Of}}{n_O}\right) \frac{n_f}{n_F}, \tag{7}$$

where n_{Of} is the number of operations in V_f from equation 6, n_O is the total number of operations in G, and n_f is the size of variable f. We make the approximation that `CustomOperations` and `ImplicitOperations` are accumulated through vector-matrix products. We can see that five out of seven problems save 60% more vector-matrix products than operation-level MAUD. For the NASA lift-plus-cruise case, 83 linear systems were solved compared to the 1083 required by operation-level MAUD. However, we note that there are cases where the condition above applies and most operations do get accumulated. These are often optimization problems where there is a single objective function that will naturally depend on all design variables. An argument can be made that variables that are not design variables should not be included in the computational graph as they can be pre-computed. This will decrease n_O in equation 7 without decreasing n_{Of}, leading to reduced impact from our method. However, it is not always known what variables will be design variables in advance. Therefore, it is advantageous for the framework to permit any variable to be design variables without loss in computational efficiency.

VI. Conclusion

In this paper, we presented a method to efficiently compute derivatives that enables efficient modeling of multidisciplinary optimization (MDO) models. We described an algorithm that generates efficient code in the last stage of the static, three-stage compiler framework described by Gandarillas et al. [3] The derivative computation is based on reverse-mode differentiation and leverages the implicit function theorem for efficient derivative propagation through nonlinear solvers. Further, we leverage the static computational graph to generate hybrid dense-sparse Jacobian accumulation code and avoid redundant vector-matrix products. We compared our method to that of an OpenMDAO-based implementation and saw performance gains of around an order of magnitude.

The most promising direction for future work consists of taking into account independent operations to automatically generate code for parallel computing environments. Further, support for higher-order derivatives is also necessary to improve optimization performance. Finally, exploring and implementing other modes of accumulation apart from reverse-mode may also improve performance.

Acknowledgments

Special thanks to Anugrah J. Joshy for implementation of operation partial derivatives. The material presented in this paper is, in part, based upon work supported by NASA under award No. 80NSSC21M0070.

References

[1] Gray, J. S., Hwang, J. T., Martins, J. R. R. A., Moore, K. T., and Naylor, B. A., "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104. https://doi.org/10.1007/s00158-019-02211-z.

[2] Gallard, F., Vanaret, C., Guénot, D., Gachelin, V., Lafage, R., Pauwels, B., Barjhoux, P.-J., and Gazaix, A., "GEMS: A Python Library for Automation of Multidisciplinary Design Optimization Process Generation," *2018 AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2018.

[3] Gandarillas, V., Joshy, A. J., Sperry, M. Z., Ivanov, A. K., and Hwang, J. T., "A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis," *Structural and Multidisciplinary Optimization*, (under review).

[4] Hwang, J. T., and Martins, J. R., "A Computational Architecture for Coupling Heterogeneous Numerical Models and Computing Coupled Derivatives," *ACM Trans. Math. Softw.*, Vol. 44, No. 4, 2018. https://doi.org/10.1145/3182393, URL https://doi.org/10.1145/3182393.

[5] Martins, J. R. R. A., and Hwang, J. T., "Review and Unification of Methods for Computing Derivatives of Multidisciplinary Computational Models," *AIAA Journal*, Vol. 51, No. 11, 2013, pp. 2582–2599. https://doi.org/10.2514/1.J052184.

[6] Griewank, A., and Walther, A., *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, Philadelphia, 2008.

[7] Margossian, C. C., "A Review of automatic differentiation and its efficient implementation," *CoRR*, Vol. abs/1811.05031, 2018. URL http://arxiv.org/abs/1811.05031.

[8] Baydin, A. G., Pearlmutter, B. A., and Radul, A. A., "Automatic differentiation in machine learning: a survey," *CoRR*, Vol. abs/1502.05767, 2015. URL http://arxiv.org/abs/1502.05767.

[9] Naumann, U., "Optimal Jacobian accumulation is NP-complete," *Math. Program.*, Vol. 112, No. 2, 2008, pp. 427–441.

[10] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A., "Automatic differentiation in pytorch," 2017.

[11] Goldsborough, P., "A Tour of TensorFlow," *CoRR*, Vol. abs/1610.01178, 2016. URL http://arxiv.org/abs/1610.01178.

[12] Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B., and Vincent, H. Y., "Chainer: A Deep Learning Framework for Accelerating the Research Cycle," *CoRR*, Vol. abs/1908.00213, 2019. URL http://arxiv.org/abs/1908.00213.

[13] Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., and Cai, S., "TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning," *CoRR*, Vol. abs/1903.01855, 2019. URL http://arxiv.org/abs/1903.01855.

[14] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T., "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[15] Maclaurin, D., *Modeling, inference and optimization with composable differentiable procedures*, 2016.

[16] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q., "JAX: composable transformations of Python+NumPy programs," , 2018. URL http://github.com/google/jax.

[17] Carpenter, B., Hoffman, M. D., Brubaker, M. A., Lee, D. D., Li, P., and Betancourt, M., "The Stan Math Library: Reverse-Mode Automatic Differentiation in C++," *CoRR*, Vol. abs/1509.07164, 2015. URL http://arxiv.org/abs/1509.07164.

[18] Al-Rfou, R., Alain, G., Almahairi, A., Angermüller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Snyder, J. B., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P. L., Cho, K., Chorowski, J., Christiano, P. F., Cooijmans, T., Côté, M., Côté, M., Courville, A. C., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Kahou, S. E., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I. J., Graham, M., Gülçehre, Ç., Hamel, P., Harlouchet, I., Heng, J., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrançois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P., Mastropietro, O., McGibbon, R., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C. J., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, É., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J. P., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y., "Theano: A Python framework for fast computation of mathematical expressions," *CoRR*, Vol. abs/1605.02688, 2016. URL http://arxiv.org/abs/1605.02688.

[19] Revels, J., Lubin, M., and Papamarkou, T., "Forward-Mode Automatic Differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016. URL https://arxiv.org/abs/1607.07892.

[20] Revels, J., "ReverseDiff.jl," , 2020. URL https://github.com/JuliaDiff/ReverseDiff.jl.

[21] Christianson, B., "Reverse accumulation and attractive fixed points," *Optimization Methods and Software*, Vol. 3, No. 4, 1994, pp. 311–326. https://doi.org/10.1080/10556789408805572.

[22] Azmy, Y., "Post-Convergence Automatic Differentiation of Iterative Schemes," *Nuclear Science and Engineering*, Vol. 125, 1997. https://doi.org/10.13182/NSE97-A24250.

[23] Blondel, M., Berthet, Q., Cuturi, M., Frostig, R., Hoyer, S., Llinares-López, F., Pedregosa, F., and Vert, J.-P., "Efficient and Modular Implicit Differentiation," *arXiv preprint arXiv:2105.15183*, 2021.

[24] Sarojini, D., Ruh, M. L., Joshy, A. J., Yan, J., Ivanov, A. K., Scotzniovsky, L., Fletcher, A. H., Orndorff, N. C., Sperry, M., Gandarillas, V. E., Asher, I., Chambers, J. T., Gill, H., Lee, S., Cheng, Z., Rodriguez, G., Zhao, S., Mi, C., Nascenzi, T., Cuatt, T., Winter, T. F., Guibert, A. T., Cronk, A., Kim, H. A., Meng, S., and Hwang, J. T., *Large-Scale Multidisciplinary Design Optimization of an eVTOL Aircraft using Comprehensive Analysis*, ???? https://doi.org/10.2514/6.2023-0146, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-0146.

[25] Orndorff, N. C., Wang, B., and Hwang, J. T., "Gradient-based sizing optimization of power-beaming-enabled aircraft," *AIAA AVIATION 2023 FORUM*, 2023.

[26] Ruh, M. L., and Hwang, J. T., "Robust modeling and optimal design of rotors using blade element momentum theory," *AIAA AVIATION 2021 FORUM*, 2021, p. 2598.

[27] Orndorff, N. C., Sarojini, D., Scotzniovsky, L., Gill, H., Lee, S., Cheng, Z., Zhao, S., Mi, C., and Hwang, J. T., *Air-taxi transition trajectory optimization with physics-based models*, ???? https://doi.org/10.2514/6.2023-0324, URL https://arc.aiaa.org/doi/abs/10.2514/6.2023-0324.