Performance portable adjoints for structured mesh applications with OPS

GÁBOR DÁNIEL BALOGH, Pázmány Péter Catholic University, Hungary
JOHANNES LOTZ, Numerical Algorithms Group, United Kingdom
JACQUES DU TOIT, Author, United Kingdom
UWE NAUMANN, Computer Science, RWTH Aachen University, Germany
ISTVÁN REGULY, Pázmány Péter Catholic University, Hungary

Derivatives are crucial for engineering and scientific applications like optimization and inverse problems. While finite difference approximations can estimate derivatives, they are computationally expensive and inaccurate. Algorithmic differentiation (AD) provides an efficient and exact method to compute derivatives by treating computer programs as mathematical functions and applying the chain rule of calculus.

Our work focuses on adjoint-mode AD for structured mesh stencil applications. We extend the Oxford Parallel Structured mesh solver library (OPS) domain-specific language to compute derivatives using the reverse mode of algorithmic differentiation. OPS allows developers to express mesh algorithms from a high-level code targeting multiple hardware from the same source. Taking advantage of the domain-specific abstraction, the extension creates a compact adjoint tape at the level of computational loops and generates the platform-specific (OpenMP and CUDA) parallel implementations for the adjoint loops, using the user provided primal and adjoint stencil-kernels.

We differentiate three example applications written in OPS, demonstrating similar performance to the original applications on both CPUs and GPUs. On these applications, computing derivatives only took $3.7 - 9.7x$ time (including the evaluation of the application) compared to the original applications, which is in line with state of the art tools.

CCS Concepts: • **Mathematics of computing** → **Automatic differentiation**; *Mathematical software performance*; • **Software and its engineering** → **Domain specific languages**; *Source code generation*; Parallel programming languages; • **Computing methodologies** → *Shared memory algorithms*; *Massively parallel algorithms.*

Additional Key Words and Phrases: Algorithmic differentiation, Structured meshes, OpenMP, CUDA, High-Performance Computing

ACM Reference Format:
Gábor Dániel Balogh, Johannes Lotz, Jacques du Toit, Uwe Naumann, and István Reguly. 2018. Performance portable adjoints for structured mesh applications with OPS. 1, 1 (September 2018), 32 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Sensitivity (mathematical derivative) information has a wide range of uses, such as gradient-based design optimization for computational fluid dynamics (CFD) applications [Albring et al. 2016; Bischof et al. 1992; Carle et al. 1994; Giles

Authors' addresses: Gábor Dániel Balogh, balogh.gabor.daniel@itk.ppke.hu, Pázmány Péter Catholic University, Práter st. 50/A, Budapest, Hungary, 1083; Johannes Lotz, johannes.lotz@nag.co.uk, Numerical Algorithms Group, 30 St Giles', Oxford, United Kingdom, OX1 3LE; Jacques du Toit, Jacques.DUToit@amd.com, Author, United Kingdom; Uwe Naumann, naumann@stce.rwth-aachen.de, Computer Science, RWTH Aachen University, Aachen, Germany, D-52056; István Reguly, reguly.istvan@itk.ppke.hu, Pázmány Péter Catholic University, Práter st. 50/A, Budapest, Hungary, 1083.

and Pierce 2000; Sanchez et al. 2018], machine learning [Baydin et al. 2018], inverse problems [Guasch et al. 2020], computational finance [Homescu 2011], image processing [Li et al. 2018], medical imaging [Grabner et al. 2008] or real-time risk management [Capriotti and Lee 2018]. Algorithmic Differentiation is an essential tool for obtaining sensitivity information in such applications: finite differences are often too expensive (or inaccurate) while deriving and then evaluating symbolic gradient information may lead to verbose and error-prone code [Laue 2022].

Algorithmic Differentiation, also known as Automatic Differentiation or AD, is a set of techniques that can compute the exact derivatives of a function with respect to its input variables by repeatedly applying the chain rule of calculus. Assume we have an application with some input variables $x \in \mathbb{R}^n$, which performs a series of steps and then computes the final result $y \in \mathbb{R}^m$. If we take this program as a mathematical function, $F : \mathbb{R}^n \to \mathbb{R}^m$, AD will compute the value

$$\bar{x} = \frac{dF}{dx}(x)^T \bar{y}.$$

Working out and manually writing code to compute analytical derivatives directly could provide the fastest solution, but it is error-prone and simply becomes infeasible for large applications. Most commonly, there are three ways to compute the derivative information automatically [Margossian 2019]: symbolic differentiation in computer algebra systems, numeric differentiation with finite differences, and algorithmic differentiation [Bücker et al. 2006; Naumann 2011]. Symbolic differentiation computes derivatives by manipulating symbolic expressions. It could provide exact derivatives, but it can be slow and requires the problem to be defined as closed-form expressions [Baydin et al. 2018], which in many cases is just not possible. Using finite differences to estimate derivatives is sometimes the easiest to implement, but the derivatives are subject to floating-point precision, and getting accurate second- or third-order information can be a real challenge. Finally, AD evaluates derivatives by differentiating the sequence of elementary arithmetic operations and elementary functions which make up a user's application, thus ensuring accuracy. Another advantage of AD over finite differences is that the computational cost related to adjoint-mode AD scales with the number of outputs instead of the number of inputs, consequently, in cases where the application produces far fewer outputs than inputs, adjoint-mode AD can compute the gradient hundreds of times faster.

Algorithmic differentiation builds on the fact that a computer program, no matter how complex, will always be constructed from a series of elementary operations such as additions, multiplications, or math functions like sin. The derivatives of these elementary operations are known and easy to compute. If we consider the function to be a composite with K elementary steps $F = f_1 \circ f_2 \circ \cdots \circ f_K$ then

$$y = F(x) = f_K(\dots f_2(f_1(x)))$$

and the derivative of y can be computed by applying the chain rule to all of the elementary functions

$$\frac{\partial y}{\partial x} = \frac{\partial f_K}{\partial f_{K-1}} \frac{\partial f_{K-1}}{\partial f_{K-2}} \cdots \frac{\partial f_1}{\partial x}. \tag{1}$$

There are two main modes of algorithmic differentiation based on which direction we start to evaluate (1): forward- (tangent-) mode AD, where the evaluation order follows the order of the original computation, and reverse- (adjoint-) mode AD, which evaluates the chain rule in reverse order. Let J denote the $m \times n$ Jacobian matrix of F, the matrix J_k is the Jacobian of the elementary function f_k, and $u \in \mathbb{R}^n$ a vector in the input space. The tangent or forward-mode AD will compute the action of the Jacobian matrix on u:

$$Ju = J_K J_{K-1} \cdots J_1 u$$

The forward-mode AD evaluates the derivatives in the same order as the original function evaluation (if F is implemented in a computer program, this is the same order in which the program evaluates statements). Choosing u to be a vector such that it is 1 for input j and 0 otherwise, a single evaluation of the tangent model will compute the j^{th} column of the Jacobian, in other words, the derivatives of all outputs with respect to a single input. Evaluation of the tangent model requires twice as much memory and typically takes roughly twice as long as the original application. However, often derivatives with respect to multiple inputs are required.

For adjoint or reverse-mode AD, let us consider $w \in \mathbb{R}^m$, a vector in the output space. Adjoint-mode AD will compute the action of the transpose of J on w:

$$J^T w = J_1^T J_2^T \cdots J_K^T w.$$

Similarly to the previous case, let us choose w such that it has a single 1 at position i and all other values are 0. A single evaluation of the adjoint model on w will produce the i^{th} row of J, or the derivatives of a single output with respect to all inputs.

The adjoint-mode AD evaluates derivatives in the reverse order, equivalent to running your computer program backwards. Doing this requires storing a Directed Acyclic Graph (DAG) of the program control flow in memory, along with local derivative information, or recomputing sections of the DAG multiple times. This can cause dramatic increases in memory requirements. The data structure that records the intermediate states and the computational graph is often referred to as the tape. One of the main challenges is to efficiently save all intermediate states and the computational steps (and their local Jacobians) while computing the result of the program, so that one can propagate derivative information from outputs back to inputs.

The choice between forward and reverse-mode AD often depends on the target application. The forward-mode can perform better for applications with a few inputs and a lot of outputs, and reverse-mode AD shines in scenarios where a function has a large number of inputs but a relatively smaller number of outputs, like gradient-based optimization and machine learning. Due to its efficiency in such contexts, adjoint-mode AD often receives special attention. Application areas include for example shape optimization tasks [Verstraete et al. 2017; Vitale et al. 2020; Özkaya and Gauger 2010], transistor modeling [Phipps et al. 2008], or in CFD solvers like the NASA Ice Sheet System Model [Larour et al. 2016], OpenFOAM [Towara and Naumann 2013; Towara et al. 2015] or STAMPS [Müller et al. 2018].

Most adjoint-mode AD tools belong to one of two categories: either they apply source transformation to generate the adjoints of the function or use operator overloading techniques [Naumann 2011]. Source transformation tools automatically transform the original code to produce a new program that calculates the adjoint values, while operator overloading tools leverage the object-oriented capabilities of C++ and other object-oriented languages to overload basic arithmetic operations and functions to compute derivatives. This allows the flexibility to easily follow complex control flow and different language constructs. Multiple tools exist that efficiently use operator overloading, like Adept [Hogan 2014], dco/c++ [Lotz 2016] or Sacado [Gay 2006; Phipps et al. 2008]. However, naively implemented operator overloading tools lead to large memory overheads for storing the tape, which is often alleviated with advanced techniques like expression templates in CoDiPack [Sagebaum et al. 2018, 2019], using direct derivatives of linear algebra constructs such as in ADiMat for MATLAB [Bischof et al. 2002], or the Stan Math library [Carpenter et al. 2015], and also using additional hybrid code generation extensions in the case of dco/c++.

While tracing the original data flow with operator overloading is easier, the resulting tape quickly increases in size. Source transformation tools can reduce the performance overhead of adjoint calculations and potentially allow more drastic optimizations at the cost of more complex tooling. Examples of adjoint-mode AD tools with source

transformation include ADIC [Narayanan et al. 2010], Tapenade [Hascoet and Pascual 2013], OpenAD [Utke et al. 2008]. In summary, algorithmic differentiation provides an effective and accurate means to compute derivatives, with adjoint-mode AD being especially pertinent in contexts with numerous input variables and only a small number of outputs. Its strategic application ensures the derivation of gradients with both computational efficiency and high precision. Our approach uses source transformation techniques that allow us to store the control flow on a higher level (at the level of entire parallel loops) while also having the ability to apply transformations and optimizations on the loops to achieve near-peak performance on parallel hardware.

We refer to the evaluation of the function F (with the additional caching and tape recording) as the forward pass. The computations that are part of F are called *primal*, and their counterparts computing the local Jacobian are called the *adjoint of the computation*. The evaluation of $J^T w$ executing the adjoints of the computational steps in reverse order we call the *reverse pass*. The *adjoint factor* is defined as the relative runtime of the adjoint computation compared to the primal. We refer to variables and computations that take part in the derivative accumulation as *active* and all other data and code as *passive*. Finally, the variables holding the values and states during the forward pass are called *primal data*, and for active data, we will refer to the variables holding derivative values as *adjoint data*.

Implementing derivatives of applications is difficult, error-prone, and particularly challenging in a parallel environment [Margossian 2019]. Ideally, users would like to be able to describe what needs to be computed in simple terms, including any derivative information, and then have a tool or framework to generate an efficient parallel implementation, including choosing the most appropriate method for obtaining the derivatives.

The parallelization of derivative computations is achieved in many ways in AD tools. A fairly simple approach for otherwise serial applications is to compute different tangent-mode AD passes, which are independent of each other, in parallel [Bücker et al. 2001; Bücker et al. 2002], or using nested parallelism for parallel applications [Bücker et al. 2004; Phipps et al. 2022]. For reverse-mode AD, there is an inherent complication. Because data flow is reversed, thread-safe concurrent reads in the primal code become thread-unsafe concurrent writes in the adjoint code. Handling these race conditions efficiently is a key concern. In [Heimbach et al. 2002], it was shown that, with source transformation tools in combination with handwritten adjoint routines, it is possible to get efficient parallel implementations for applications.

Another approach is the use of tool-agnostic extension libraries for handling parallelism, which was introduced with the AMPI library [Schanen et al. 2012, 2010] for handling MPI constructs combined with traditional AD tools, which has been shown to successfully integrate with the operator overloading library dco/c++ [Lotz et al. 2013]. OpDiLib [Blühdorn et al. 2023] is another tool-agnostic library for shared memory parallelism with OpenMP or hybrid codes with OpenMP and MPI. The other approach is to integrate special treatment of MPI constructs into the AD library, such as in TAF [Giering 2026; Giering et al. 2005, 2006].

OpenMP parallelism is controlled through `#pragmas`, which are hard to follow with conventional operator overloading tools since there is no function to overload like in the case of conventional MPI, and tools had to rely on constructors and destructors.

Multiple tools have recently developed extensions for supporting shared memory parallelism. PARAD [Kaler et al. 2021] extends the operator overloading tool Adept to support OpenMP parallel constructs, and OpDiLib [Blühdorn et al. 2023], as mentioned earlier, supports OpenMP through either wrapping the directives into macros or the event-based OMPT API. The source transformation tool Tapenade also gained support for some OpenMP directives through a theoretical model [Hückelheim and Hascoët 2022] and a more complete support for Fortran with a formal extension FormAD [Hückelheim and Hascoët 2023]. The Enzyme source transformation tool performs the code generation inside the compiler on the LLVM IR level [Moses and Churavy 2020], taking advantage of the optimization passes of the

compiler prior to the code generation and using rule-based transformations for OpenMP and MPI [Moses et al. 2022]. PerforAD showed the capability to generate race-free OpenMP for adjoint-based loop transformations for stencil loops [Hückelheim et al. 2019]. For stencil codes in particular, an efficient CPU-based parallelization strategy was developed in [Hückelheim et al. 2018], for forward-mode AD.

Similarly to the Stan Math Library, using specialized adjoint functions for operations like matrix-matrix multiplication, adjoint-mode AD can be applied to CUDA applications efficiently [Gremse et al. 2016]. A handful of tools provide support for CUDA as well. AutoMat [Blühdorn et al. 2022] shows possibilities for operator overloading for CUDA kernels but heavily depends on recomputing values and lacks high-level checkpointing strategies like Revolve [Griewank and Walther 2000]. Enzyme showed the ability to generate adjoint kernels for race-free CUDA kernels using low-level analysis of access patterns in the IR, avoiding superfluous barriers if possible and using atomics only on memory location accessible by other threads [Moses et al. 2021].

On modern hardware, the parallel execution of algorithms is necessary to solve larger and more complicated problems. There are many frameworks to write parallel algorithms with different advantages. MPI, OpenMP, and their combination have been the most common ways to target CPUs for a long time. By contrast, the landscape for available frameworks for GPUs is much more complicated, with vendor-specific frameworks providing the highest performance. Maintaining a code base to target all this hardware for an application is infeasible. To solve this problem, general purpose frameworks have arisen in recent years to target all platforms from a single source, such as SYCL [Reyes and Lomüller 2016], KOKKOS [Edwards et al. 2014; Trott et al. 2022] or RAJA [Beckingsale et al. 2019]. However, due to the differences in the hardware, relatively low-level approaches like SYCL do enable functional portability, but struggle to achieve meaningful performance portability without a fair amount of platform-specific tuning. The other approach is to focus on performance portability using higher-level abstractions, such as in KOKKOS and RAJA, and sacrifice generality in terms of ways to express computations. Another example of trading generality for performance portability is the use of domain-specific languages (DSL) or abstractions. By sacrificing generality in terms of application areas, DSLs can express algorithms using higher-level concepts specific to that domain, concepts that have a natural meaning to the application developer. The DSL can take advantage of this meaning and any accompanying restrictions to perform a wide range of optimizations to generate efficient code for the target hardware.

This paper focuses on adjoint-mode differentiation of stencil computations in the Oxford Parallel library for Structured mesh solvers (OPS)[1] [Reguly et al. 2018] embedded Domain Specific Language (eDSL) designed to bridge the gap between high-level scientific computations and the optimal use of parallel and distributed systems. Integrated seamlessly into C, C++, and Fortran, OPS empowers computational scientists and engineers by providing a suite of abstractions that cater to structured mesh computations.

The Devito [Louboutin et al. 2019] DSL embedded in Python is designed to provide an abstraction over finite-difference partial differential equation solvers. It starts from high-level symbolic expressions of the problem and using a stencil compiler to generate the parallel implementations for the application. Another important DSL is Halide [Li et al. 2018] embedded in C++ that provides an abstraction to express image and array processing tasks. From their high-level representation these tools can perform a wider range of transformations e.g. Devito can formulate the adjoint equations.

OPS uses a slightly lower-level abstraction over structured mesh computations operating on the level of computational loops. Computations in OPS are represented as a series of stencil loops applying a specific operation to each element of a structured grid accessing a fixed pattern of neighboring elements, called a stencil. In other words, a stencil specifies

[1] https://op-dsl.github.io/

how to combine the values of neighboring cells in a grid to compute a new value for a given cell. Stencil computations are widely used in many fields, including CFD [Becker et al. 2010], climate modeling [Heimbach et al. 2005; Singh et al. 2023] and computational finance [Clevenhaus et al. 2021; Düring et al. 2014; Mollapourasl et al. 2020; Wang et al. 2011]. They are efficient and scalable, making them suitable for large-scale simulations on parallel computing architectures. The parallelization of stencil codes is reasonably straightforward, and their performance and optimizations to perform stencil loops on modern parallel architectures efficiently are widely studied [Lakshminarasimhan et al. 2024; Mudalige et al. 2019; Rahman et al. 2011; Raut et al. 2020; Reguly et al. 2021]. The structure of the stencil loops nicely combines with code generation approaches like stencil compilers. YASK [Yount et al. 2016], HOSTS [Stock et al. 2014], or Tiramisu [Baghdadi et al. 2019] can automatically perform extensive optimizations and parallelization from higher level stencil abstraction that describe the stencil computation, and different scheduling patterns. While OPS takes a similar approach it is using a lower-level abstraction that makes OPS more easily adaptable for a wider range of numerical algorithms, as well as already existing code bases.

By separating the high-level application code from the low-level parallel implementation, OPS removes the need for scientists and engineers to grapple with the nitty-gritty details of parallelism and data movement, like scheduling and platform specific programming models, allowing them to write a single high-level source code. At the same time, the library takes care of the optimizations for the target architectures.

OPS defines several key abstractions for defining computations. *Blocks* are N dimensional containers on which datasets are defined. *Datasets* are collections of data entities associated with blocks and serve as primary operands for *computational kernels* and accessed through predefined *stencils*. The computations within OPS are typically expressed with the parallel loop construct, where user-defined kernels operate over specific iteration ranges and datasets defined on the same block. During the automatic transformation step, OPS will replace these loop constructs with calls to the target-specific parallel implementations. The loops require the user to specify how each dataset is accessed by the user kernel – such as whether data is to be read, written, or incremented, and the specific stencil pattern governing data access. OPS requires the parallel operation to be insensitive to the order of execution on individual grid points (within machine precision). Also, the kernel must not have any side effects on the program state outside of the changes on the datasets, and reductions marked in the argument of the parallel loop. Using this abstraction, the domain scientist can express structured mesh algorithms without prescribing how to execute them in a parallel environment with heterogeneous memory spaces.

Listing 1 shows the API for creating a dataset on a block and then using the datasets in an OPS parallel loop from the Poisson example application. The scientists express the computation in the form of these computational loops, providing the loop body that will be applied for each grid point, the block and the iteration range, and a descriptor for each argument of the loop body. These descriptors contain the information of the dataset, the access pattern, the stencil, and the underlying data type. There are some additional rules that each parallel loop and the user defined kernel function must follow. OPS requires all data used by the kernels to be registered to the backend before using them in the kernels. These data can be defined on each grid point, constants in global scope, and reductions. The computational kernels must not use global variables other than the global constant registered by `ops_decl_const` prior to the kernel call, which will register the symbol in OPS to ensure that the symbol will be available on all used devices. Grid invariant constants can be passed as `ops_arg_global` to be used by the kernels. Data access to `ops_dats` inside the kernel happens through accessor objects that allows multidimensional relative indexing of the datasets. The source-to-source translation layer of OPS takes this information and produces the hardware-specific low-level parallel implementations for the kernel. OPS supports and generates code for a range of target hardware with different parallel

```
double dx, dy;

// User kernel
void update(const ACC<double> &u, const ACC<double> &f,
            ACC<double> &u2) {
  u2(0, 0) = ((u(-1, 0) + u(1, 0)) * dy * dy +
              (u(0, -1) + u(0, 1)) * dx * dx -
              f(0, 0) * dx * dx * dy * dy) /
             (2.0 * (dx * dx + dy * dy));
}
// ...
ops_decl_const("dx", 1, "double", &dx); // Register global
ops_decl_const("dy", 1, "double", &dy); // constants
// Declaring the computational domain
ops_block block = ops_decl_block(2, "grid");
// Declaring a dataset on a block
int size[2] = {size_x, size_y}
int base[2] = {0, 0};
// max halo depths for the dat in all direction
int d_p[2] = {1, 1};
int d_m[2] = {-1, -1};
ops_dat u = ops_decl_dat(block, 1, size, base, d_m, d_p,
                         nullptr, "double", "u");
// Declaring f and u2
// Declaring access pattern descriptors
int s2D_00[]         = {0,0};
ops_stencil S2D_00 = ops_decl_stencil(2, 1, s2D_00, "00");
// S2D_4PT
// Execute a given loop on the block
int iter_range[] = {0, size_x, 0, size_y};
ops_par_loop(update, "update", block, 2, iter_range,
         ops_arg_dat(u, 1, S2D_4PT, "double", OPS_READ),
         ops_arg_dat(f, 1, S2D_00, "double", OPS_READ),
         ops_arg_dat(u2, 1, S2D_00, "double", OPS_WRITE));
```

Listing 1. Example declaration of an OPS dataset, global constants and the required stencils with their use in an OPS parallel loop.

programming models such as OpenMP, OpenACC, CUDA, and their combination with MPI. The user is not required to modify the OPS application source to use these paradigms. The transformed source code is then compiled using traditional compilers and linked against OPS backend libraries.

Note that the ownership of all datasets is handed to the library and can only be accessed through OPS APIs, which allows OPS to keep track of the state of all datasets and when it is required to move the data, for example, between CPUs and GPUs. The ability to follow data movement and dependencies throughout the application has already shown to be a critical factor in allowing OPS to perform complex optimizations like loop tiling and lazy execution [Reguly et al. 2018] and also plays a crucial role in providing reverse-mode AD in OPS.

OPS aids researchers in their productivity by allowing them to develop a single high-level application source and then automatically access all optimizations OPS provides for past and future architectures. Transitioning to the OPS abstraction naturally comes with costs, such as refactoring loops to outlined loop bodies and `ops_par_loop` calls. Nevertheless, by maintaining a unified codebase, the application can seamlessly gain access to highly efficient

implementations for a wide range of hardware without developing any platform specific optimizations or application code.

OPS is responsible for scheduling the application and its data, which gives additional opportunities for optimizations both at a loop level and on the level of loop chains.

Extending a DSL like OPS with AD support enables a new class of applications to take full advantage of the performance-portability and optimizations provided by the DSL. To enable AD for an OPS application, the original application code requires minimal changes (such as seeding and accessing derivatives) and the adjoints of the loop bodies. The user defines the application in the abstraction of OPS, provides the loop bodies and their adjoints as functions. The OPS library itself can generate code based on the additional information and restrictions of the domain, which enables diagnostics and optimizations that would require extensive code analysis or are just impossible for general-purpose tools. The descriptive nature of the OPS API combined with the source generation gives fine control over the parallelism in the loops and the derivatives. Many of the tools mentioned above depend on large tapes to follow the control flow of the applications. In OPS, the code transformation and optimization step for the parallel implementations happens at compile time with additional augmentation to build the tape. This enables OPS to reduce the adjoint AD memory overhead by using a high-level representation of the computational steps. At runtime, OPS builds a highly compact tape of the entire simulation where each entry in the tape represents a parallel loop or an external function (see Section 2.3).

This paper follows on from earlier work [Balogh and Reguly 2020] that offered a glimpse into OPS's adjoint-mode AD with OpenMP and expands it by thoroughly describing the extensions to the OPS abstraction and the parallel implementations targeting CPU and GPU architectures.

Overall, our paper makes the following contributions:

- We present a model for reverse-mode differentiation of complex stencil applications using OpenMP and CUDA from the same source expressed in the OPS domain-specific language[2]. We introduce the mapping of the high-level description of computational loops to the parallel implementation of the adjoint loops. We show that extensive transformations and optimizations on the parallel implementation of the adjoint loops are feasible with the information provided by the DSL.
- We introduce an AD tape to the OPS DSL with elements handled on the computational kernel level, allowing more compact storage thanks to the OPS abstraction. Our implementation shows that this tape makes the implementation of extensions, such as optimal AD checkpointing via Revolve [Griewank and Walther 2000] or arbitrary "external" (to OPS) adjoint functions, such as linear solvers, reasonably straightforward.
- Experimental results for three applications implemented in OPS are shown: a simple Poisson equation solver, the Cloverleaf hydrodynamics mini-application [Mallinson et al. 2013], and a Convection Diffusion Reaction equation (CDE) solver (implementing the Stochastic Local Volatility model from computational finance) [Wyns and Du Toit 2017]. We present detailed performance of the OpenMP and CUDA implementations. We show performance results for two implementations to compute adjoints for the Poisson application, one with traditional implementation and one with the application expressed as a fixed point iteration. Additionally, we show the performance of Revolve as the checkpointing algorithm for Cloverleaf and the CDE solver.

[2]Source is available at https://github.com/OP-DSL/OPS/tree/feature/ad2

2 REVERSE-MODE ALGORITHMIC DIFFERENTIATION IN OPS

Our extension for OPS uses reverse-mode AD and does not support forward-mode AD; from this point on, the discussion focuses on reverse-mode AD.

OPS follows the control flow through the `ops_par_loop` calls (such as in Listing 1). These calls refer to computational kernels, listing all accessed datasets and the access patterns. This makes them the perfect candidate to become the building block of the control flow graph in OPS. Since OPS has all the required information on the data dependencies on a loop level, we can build a DAG on a computational loop level instead of on an expression level, resulting in a smaller memory footprint. OPS will handle the execution of the adjoint loops by caching intermediate states during the primal evaluation and re-loading these states in the adjoint loops in reverse order.

```
// Adjoint user kernel
void update_adjoint(const ACC<double> &u, ACC<double> &u_a1s,
                    const ACC<double> &f, ACC<double> &f_a1s,
                    ACC<double> &u2, ACC<double> &u2_a1s) {
  double div = (2.0 * (dx * dx + dy * dy));
  u_a1s(-1, 0) += u2_a1s(0, 0) * dy * dy / div;
  u_a1s(1, 0) += u2_a1s(0, 0) * dy * dy / div;
  u_a1s(0, -1) += u2_a1s(0, 0) * dx * dx / div;
  u_a1s(0, 1) += u2_a1s(0, 0) * dx * dx / div;
  f_a1s(0, 0) += u2_a1s(0, 0) * dx * dx * dy * dy / div;
  u2_a1s(0, 0) = 0;
}
```

Listing 2. The user given adjoint loop body for the kernel from Listing 1.

The only component required to do adjoint-mode AD is the adjoint functions of the kernel bodies. OPS already generates the primal kernel from the kernel function and the data presented by the stencils. If OPS has a reference to the adjoint of the kernel body (such as in Listing 2) function, it can use the same information to generate parallel loops for the adjoint of the kernel.

In an OPS loop, each dataset is accessed through a stencil with an assigned access pattern: read, write, or increment. All forward loops in OPS must use gather stencils only, meaning that read can appear with any stencil with any number of points, but increment and write stencils must have one single point access with zero offsets. In reverse-mode AD in the adjoint loops, the data flow will be reversed from gathering stencils, and we will get scatter operations on the adjoint data. Due to the reversal, the write and increment stencils on the datasets will turn into one-point read stencils on the adjoint data, and read stencils will turn into increment stencils with multiple points. Figure 1 shows the change in the access pattern between the primal and adjoint loop for a five-point stencil. The above has two implications: first, the primal loops are race-free, the parallelization is trivial, and second, the adjoint loops will have data races on the adjoint data. To handle these data races we need information about the structure of these data races. We need to know which adjoints will be incremented by the adjoint loop and the pattern or location of the increments performed on the adjoints. However, OPS already knows the access pattern of all data based on the parallel loop descriptor and the access patterns on the adjoints follow the reversed pattern, hence OPS has all the information on where these data races occur from the read stencils of the primal loops.

OPS has two other loop parameter types: grid invariant constants and global reductions. We keep the global constants as passive data, which will not have derivatives, and introduce a new `ops_scalar` type for global read parameters with

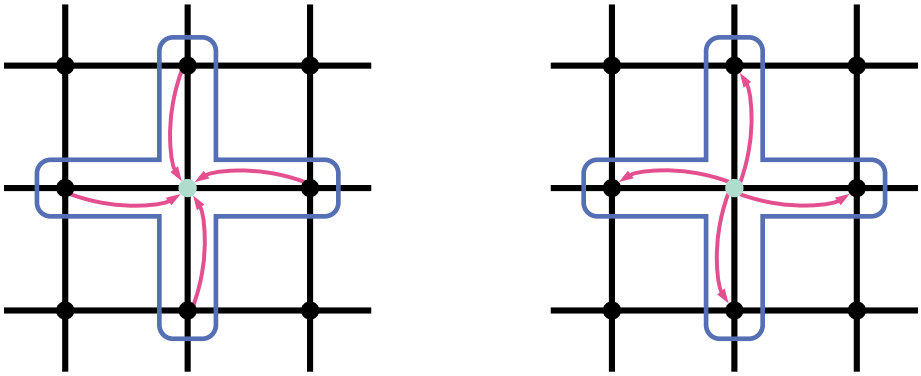


Fig. 1. **Left:** The gather stencil used during the forward pass in the parallel loop in Listing 1. Data is read on the four neighboring points and in the center and only writes at the center. **Right:** The stencil and access pattern used during the reverse pass on the adjoint memory with the loop body in Listing 2. The data is read and written on the five points.

adjoint data. Reads on `ops_scalars` will turn into global reductions in the adjoint loops. Finally, global reductions become global reads on the adjoint data in the adjoint loops.

```
// User kernel
void kernel(const ACC<double> &a, const double* scalar, ACC<double> &anext) {
  anext(0, 0) += *scalar * (a(1, 0) - a(0, 0) + a(-1, 0));
}
// Adjoint User kernel
void kernel_adjoint(const ACC<double> &a, ACC_A1S<double> &a_a1s,
                    const double* scalar,         double* scalar_a1s,
                    ACC<double>   &anext, ACC_A1S<double> &anext_a1s) {
  a_a1s( 1, 0) +=      *scalar * anext_a1s(0, 0);
  a_a1s( 0, 0) += -1 * *scalar * anext_a1s(0, 0);
  a_a1s(-1, 0) +=      *scalar * anext_a1s(0, 0);
  *scalar_a1s  += (a(1, 0) - a(0, 0) + a(-1, 0)) * anext_a1s(0, 0);
}
// ...
// Declaring a dataset on a block
double scalar = 1.0;
ops_scalar scl = ops_decl_scalar("scl", &scalar, 1);
// Registering and creating a, anext, S2D_3PT, S2D_00
// Execute a given loop on the block
ops_par_loop(kernel, "kernel", block, 2, iter_range,
             ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
             ops_arg_scalar(scl, 1,          "double", OPS_READ),
             ops_arg_dat(anext,  1, S2D_00,  "double", OPS_INC));
```

Listing 3. Example parallel loop using active grid invariant scalar data. OPS uses the `ops_scalar` datatype to handle the lifetime of the grid invariant data and it's adjoint.

Listing 3 shows a kernel with an active scalar value. To use `ops_scalars` in the kernel, it must be passed with a `ops_arg_scalar` wrapper with `OPS_READ` access. This argument will notify OPS to generate the proper code for the adjoints as well. Note that the adjoint loop has a global reduction in the derivative of the scalar, which will naturally have a significant effect on the performance during the execution of the adjoint loop. On the other hand, the opposite

happens for reductions in primal loops, which can be removed entirely from adjoint loops and produce a global read on the adjoint memory.

2.1 Adjoint function of the loop body

OPS handles the code generation of the adjoint loops similarly to the primal, which requires a function pointer to the loop body. OPS depends on the function provided by the user as an adjoint of a loop body with the `_adjoint` suffix. These functions generally follow the same rules as the user functions for the primal loops with a few exceptions. Arguments with `_a1s` suffixes are used as the adjoint of the arguments, and incrementing their values with non-zero offset is allowed. To help provide these functions, we implemented a helper script to generate adjoints of user kernels using Tapenade [Hascoet and Pascual 2013], which can help to generate adjoints for relatively simple loop bodies and can give a good starting point to write the adjoints for more complex functions. This tool acts as a wrapper for calling Tapenade on the user kernels, mapping the OPS API and some C++ constructs to a format for which Tapenade can generate adjoints. The tool treats `ops_dat` and `ops_scalar` parameters as active at the moment. After generating the adjoint functions, OPS uses them as the user-provided adjoints for the loops to generate the backend-specific parallel loops. In our experiments, we used this tool to generate the adjoint implementations for the 87 distinct kernels in Cloverleaf. A few kernel required further modifications to make the adjoint kernels compatible with CUDA. In these cases we only needed local changes to remove the stack objects that Tapenade uses to handle branches, but lengthy, complicated kernels could require significant coding effort to get an efficient adjoint function compatible with OPS.

2.2 Getting derivatives in OPS

Listing 4 shows an example AD workflow in OPS. The code computes the adjoints with respect to a given output in five main steps in OPS. Lines 1 - 20 initialize the datasets and grid invariant values (ops_scalars) before the primal, followed by actually computing the primal the same way as a traditional OPS application in lines 12-21. Then in line 27, seed the output variable's derivative, and line 30 runs the reverse pass. Finally getting a copy of the derivatives in line 37. During the primal execution, OPS will save all events (loops, reductions, etc.) relevant for derivative propagation into its own tape and save the intermediate states if necessary. Figure 2 shows how OPS handles a parallel loop call during the forward pass. OPS saves a small descriptor of the parallel loop in memory to create a representation of the computation. Based on this descriptor, OPS will execute the primal loop and store the overwritten primal data. After the primal execution, the user seeds the adjoints of the output (see line 27 of Listing 4), and finally, the `ops_interpret_adjoints` call will run the adjoint of each computational step in reverse order, accumulating derivative information. This final step will allocate and initialize all required adjoint variables that are not seeded before the function call. For each event in the DAG, OPS will execute the corresponding adjoint function, such as calling the adjoint loop as shown in Figure 2. Listing 4 also shows the two main ways to access derivatives of datasets. The API is similar to the data access API in OPS. The user can either get access to the raw pointer underneath the `ops_dat` variable or can create a copy in some memory space.

In order to maximize performance and constrain memory use, it is helpful to identify which variables are taking part in the propagation of derivative information. Existing tools use different conventions - for example, most operator overloading tools embed this information into their types. Some source transformation tools like Tapenade treat all pointers to floating point values as active, and all variables copied by value as passive constants or depend on an explicit listing of active input and outputs. In OPS, all datasets declared with `ops_decl_dat` are considered active, the user can define passive datasets with the `ops_decl_dat_passive` function to avoid the allocation of the adjoint variables, and

```
// Step 1: OPS initialisation
std::unique_ptr<OPS_instance> instance = std::make_unique<OPS_instance>(argc, argv, 1);
// Mesh
ops_block block = instance->decl_block(2, "The Block");
// Preparing data for ops
ops_scalar scl = ops_decl_scalar("scl", &scalar, 1);
ops_dat a_in = block->decl_dat(1, size, base, pad_p, pad_m, input_a, "double", "a_in");
ops_dat a    = block->decl_dat(1, size, base, pad_p, pad_m, nullptr, "double", "a");
ops_dat a2   = block->decl_dat(1, size, base, pad_p, pad_m, nullptr, "double", "a2");
// ...
// Step 2: Run the code to be differentiated
int iter_range[] = {0, size_x, 0, size_y};
ops_par_loop(copy, "copy", block, 2, iter_range,
             ops_arg_dat(a_in,   1, S2D_00, "double", OPS_READ),
             ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
for (int i = 0; i < niter; ++i) {
  ops_par_loop(kernel, "kernel", block, 2, iter_range,
               ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
               ops_arg_scalar(scl, 1,          "double", OPS_READ),
               ops_arg_dat(a2,     1, S2D_00,  "double", OPS_INC));
  std::swap(a2, a);
}

// Step 3: Initialise and seed Adjoints
int memspace = OPS_HOST;
auto *a_a1s = reinterpret_cast<double *>(a->derivative_get_raw_pointer(S2D_00, &memspace));
// Seed the output at the point of interest
a_a1s[output_idx] = 1.;
a->release_raw_derivative(OPS_WRITE, memspace);
// Step 4: Interpret Adjoints
ops_interpret_adjoints(instance.get()));

// Step 5: Accessing derivatives
int disp[OPS_MAX_DIM]; int size[OPS_MAX_DIM];
ops_dat_get_extents(a_in, 0, disp, size);
size_t array_size = size[0] * size[1];
std::vector<double> output(array_size);
a_in->fetch_derivative(output.data(), OPS_HOST);
```

Listing 4. Typical steps for an Adjoint workflow in OPS.

similarly, the user can create passive loops with `ops_par_loop_passive` function for loops that are not part of the derivative propagation. For active loops, OPS can detect active variables from the signature of the adjoint function of the loop body. The example kernel in Listing 3 shows a loop where all inputs are active, and their derivative is present in the argument list of the adjoint. By simply leaving out the derivative of an input from the signature of the adjoint of a given kernel function, OPS will treat the relevant data in that kernel as passive.

2.3 Computations outside of OPS

In some cases, pure stencil computations cannot give sufficient performance, or methods that cannot be represented in the abstraction of OPS are required. An important case here is direct linear solvers, which are used to implement High-Order Compact discretization schemes or modern direction splitting time steppers such as Douglas, Modified

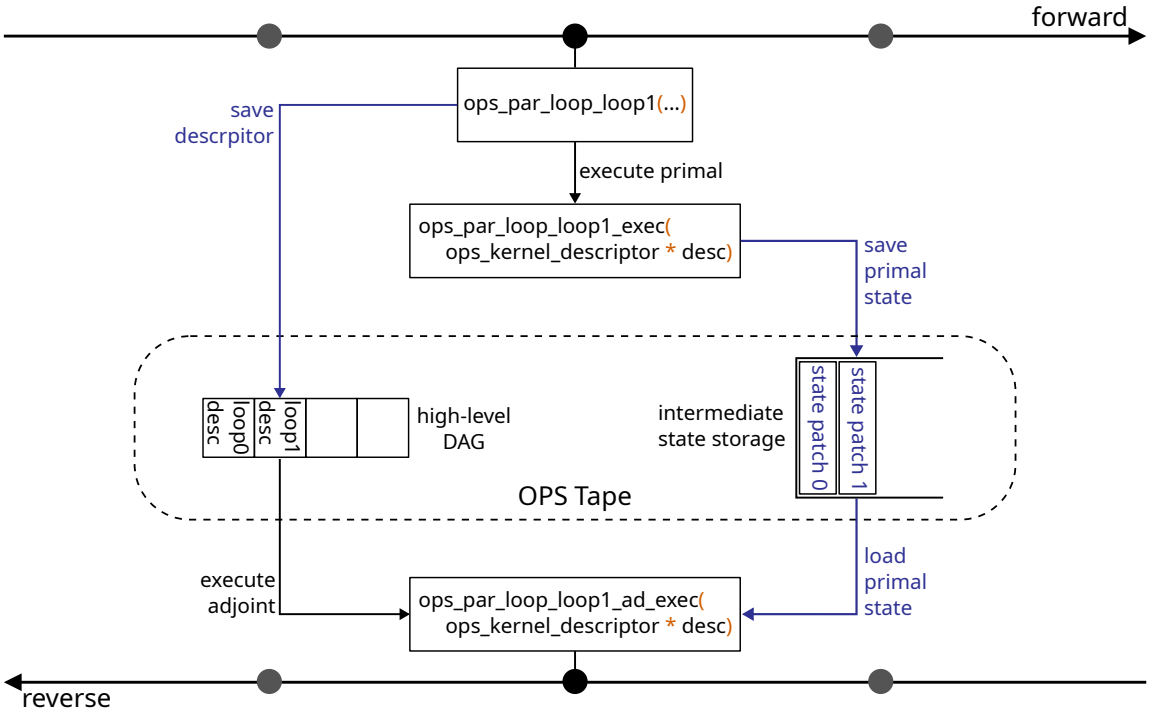


Fig. 2. Interactions of forward and corresponding reverse loops with the high-level tape in OPS. The **black arrows** indicate the control flow related to the parallel loop, while **blue arrows** represent corresponding data movements. Each active forward loop pushes a small descriptor into the DAG, and then execute the primal loop. During execution, it will save the overwritten data to a stack-like storage. In the reverse pass, OPS will call the generated adjoint for the loop, which will propagate the derivative information and load the saved state.

Craig-Sneyd, or Hundsdorfer-Verwer [Hundsdorfer et al. 2003]. The standard solution in a non-AD context is to ask for access to the raw data behind the OPS datasets, perform the computations outside of OPS, and return the data to OPS. We refer to such calculations which take place outside of the OPS abstraction as *external calculations*. OPS provides an external adjoint API to support this feature in an AD-aware manner, where during OPS stops recording the tape and the user needs to provide a function that will fill the gap in the tape. For these external computations OPS takes a function pointer to the primal, an explicit list of datasets accessed in the external function, similar to an OPS loop, and a function pointer to the adjoint of the external function provided by the user. Inside these functions, the user can execute arbitrary computations with the traditional APIs and even OPS loops as long as all data ownership is returned to OPS before the end of the function. It is crucial that the function does not modify any global program state not owned by OPS since this could result in incorrect values should checkpointing be used. For such cases, to avoid wrong incorrect results during recomputation, OPS provides a `ops_execute_external_function` version which always saves the affected OPS data and will load these states instead of re-executing the external function at the cost of the additional memory overhead. Listing 5 shows an external function from the CDE application accessing four datasets that form a batch of tridiagonal systems, solve the systems, and return ownership to OPS. The ad function contains the adjoint of the external function, which consists of a tridiagonal solve of the transposed system and an OPS loop on the derivatives.

```
auto primal = [=]() {
  // get access to used ops_dats
  double *a_raw = reinterpret_cast<double *>(ops_dat_get_raw_pointer(a, 0, S2D_00, &mem_space));
  // ...
  tridDmtsvStridedBatch(&trid_params, a_raw, b_raw, c_raw, d_raw,
                        ndim, solvedim, dims, stride);
  ops_dat_release_raw_data(a, 0, OPS_READ); // ...
};
auto ad = [=]() {
  // get access to used ops_dats
  double *a_raw = reinterpret_cast<double *>(ops_dat_get_raw_pointer(a, 0, S2D_00, &mem_space));
  double *d_a1s = reinterpret_cast<T *>(ops_dat_derivative_get_raw_pointer(d, S2D_00, &mem_space));
  // ...
  tridDmtsvStridedBatch(&trid_params, c_raw - 1, b_raw, a_raw + 1, d_a1s,
                        ndim, solvedim, dims, stride);
  // Release datasets
  ops_dat_release_raw_derivative(d, OPS_RW, mem_space);
  ops_dat_release_raw_data(a, 0, OPS_READ);
  // ...
  // Update a1_M =  -a1_d * x^T
  ops_dat a1_a = ops_get_derivative_as_ops_dat(a);
  // ...
  ops_par_loop_passive(update_a1_M, "update_a1_M", theBlock, 2, range, ops_arg_idx(),
                       ops_arg_gbl(&nx, 1, "int", OPS_READ),
                       ops_arg_dat(d, 1, S2D_3PT_X, "double", OPS_READ),
                       ops_arg_dat(a1_d, 1, S2D_00, "double", OPS_READ),
                       ops_arg_dat(a1_a, 1, S2D_00, "double", OPS_INC),
                       ops_arg_dat(a1_b, 1, S2D_00, "double", OPS_INC),
                       ops_arg_dat(a1_c, 1, S2D_00, "double", OPS_INC));
};
ops_execute_external_function_recomp(primal, ad,
                                     ops_arg_dat(a, 1, S2D_00, "double", OPS_READ),
                                     ops_arg_dat(b, 1, S2D_00, "double", OPS_READ),
                                     ops_arg_dat(c, 1, S2D_00, "double", OPS_READ),
                                     ops_arg_dat(d, 1, S2D_00, "double", OPS_RW));
```

Listing 5. An external function solving a batch tridiagonal system formed by the datasets. The `primal` function will be called by the `ops_execute_external_function_recomp` function during the forward pass, while the `ad` function provides the adjoint for the external function.

2.4 Controlling memory requirements

Adjoint-mode AD executes the adjoint of each instruction of the primal in reverse order, for which it requires the intermediate states of the variables. There are two ways to produce these intermediate states that are combined in most tools: saving the states during the primal and reloading from memory and recomputing. OPS, by default, saves all data that a loop writes at the beginning of every loop. During the reverse pass, OPS will load these, restoring all variables to the state before the primal loop. With this strategy, OPS does not need to rerun primal loops, which means faster overall run-time, but this introduces significant memory requirements to checkpoint all these intermediate states. Existing tools use checkpointing strategies to manage the trade-off between extra computational steps due to recomputing and the high memory requirement of adjoint-mode AD. OPS uses an adapted implementation of the Revolve [Griewank and Walther 2000] algorithm, originally developed for time marching schemes.

```
// Initialise Revolve
ops_ad_set_checkpoint_count(instance.get(), 10, niter);
// Register a Revolve checkpoint
ops_ad_manual_checkpoint_datlist(a_in);
// Run the code to be differentiated
ops_par_loop(copy, "copy", block, 2, iter_range,
             ops_arg_dat(a_in,   1, S2D_00, "double", OPS_READ),
             ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
for (int i = 0; i < niter; ++i) {
  // Register a Revolve checkpoint
  ops_ad_manual_checkpoint_datlist(a, a2);
  ops_par_loop(kernel, "kernel", block, 2, iter_range,
               ops_arg_dat(a,      1, S2D_3PT, "double", OPS_READ),
               ops_arg_scalar(scl, 1,          "double", OPS_READ),
               ops_arg_dat(a2,     1, S2D_00,  "double", OPS_INC));

  ops_par_loop(copy, "copy", block, 2, iter_range,
               ops_arg_dat(a2,     1, S2D_00, "double", OPS_READ),
               ops_arg_dat(a,      1, S2D_00, "double", OPS_WRITE));
}
```

Listing 6. Example use of Revolve for the loop chain from Listing 4 with 10 checkpoints and a fixed chain length of `niter`. The `ops_ad_set_checkpoint_count` call initiates Revolve, and the `ops_ad_manual_checkpoint_datlist` function marks the location of the checkpoints from which OPS can start recomputing steps. The function takes all the `ops_dats` that are required to recompute the iteration.

OPS enables users to define checkpoint locations, manually specifying which datasets to save. OPS will store the information of all possible checkpoints in the DAG and use Revolve to manage rerunning loops and saving intermediate states for these checkpoints. Listing 6 shows the Revolve API for OPS. At the beginning of the application, OPS requires the user to set the parameters for Revolve with `ops_ad_set_checkpoint_count`, and then Revolve decides which checkpoints should we store data for in the chain. At the beginning of each application-level iteration, the user provides a list of datasets to build a checkpoint with `ops_ad_manual_checkpoint_datlist`. The list should contain datasets that are required to reset their states to this point to compute the current iteration, or in other words, all datasets that are read and will be overwritten later in the application. Every time OPS encounters a `ops_ad_manual_checkpoint_datlist` call, it will save the list of datasets provided, and if the datasets should be stored for the checkpoint, then stores a copy of each dataset to memory and compute the index of the next checkpoint, that will be stored. In the reverse pass, OPS will interpret the adjoint of one iteration at a time. For each iteration, it will recompute iterations from the last stored checkpoint and cache the intermediate states for the last iteration to prepare it for the reverse pass. If OPS passes a stored checkpoint in the reverse pass, OPS frees the stored data, and when OPS recomputes sections of the DAG and has available slots for checkpoints, OPS will store datasets for checkpoints based on the Revolve algorithm.

2.5 Retaping and reverse accumulation

OPS provides access to its tape object to give a finer level of control if needed. Reusing the tape multiple times allows the user to create and compose a hierarchy of tapes that allows advanced techniques like reverse accumulation for fixed point iterations [Christianson 1994] that can drastically reduce the size of the tape and cached states.

Listing 7 shows an example of reverse accumulation. OPS can create a local tape in the adjoint, record computations, and reuse the computational graph to form a convergent iteration using the local tape to compute the adjoint for the

```
auto primal = [&]() {
  double error = std::numeric_limits<double>::max();
  while (error > tolerance) {
    ops_par_loop(step, "step", block, 2, range, // ...
                 ops_arg_dat(a, 1, S_00, "double", OPS_INC),
                 ops_arg_reduce(err, 1, "double", OPS_MAX));
    err->get_result(&error);
  }
};
auto ad = [&]() {
  ops_dat derivative = a->get_derivative_as_ops_dat();
  ops_reduction maxerr = ops_decl_reduction_handle(sizeof(double), "double", "maxerr");
  ops_ad_tape *tape = ops_create_tape(OPS_instance::getOPSInstance());
  // record one iteration
  ops_par_loop(step, "step", block, 2, range, // ...
               ops_arg_dat(a, 1, S_00, "double", OPS_INC),
               ops_arg_reduce(tmperr, 1, "double", OPS_MAX));
  double error = std::numeric_limits<double>::max();
  while (error > tolerance) {
    // interpret adjoints
    tape->interpret_adjoint(OPS_instance::getOPSInstance());
    // check error
    ops_par_loop_passive(maxerr, "maxerr", block, 2, range, // ...
                         ops_arg_dat(derivative, 1, S_00, "double", OPS_READ),
                         ops_arg_reduce(maxerr, 1, "double", OPS_MAX));
    maxerr->get_result(&error);
    // zero adjoints in tape
    tape->zero_adjoints_except({a});
    // reseed others like scalars
    // ...
  }
  ops_remove_tape(OPS_instance::getOPSInstance(), tape);
};
ops_execute_external_function(primal, ad, // ...
                              ops_arg_reduce(err, 1, "double", OPS_MAX),
                              ops_arg_dat(a, 1, S_00, "double", OPS_INC));
```

Listing 7. Example code for executing a fixed point iteration in the primal with using reverse accumulation to compute the adjoints. In the adjoint of an external function, the user can create local tapes to use. Similarly to other external functions, the user can use passive loops for error calculation, reseeding, or other purposes.

whole application. Since all iterations in the primal are performed in the external function (or performed as passive loops), these loops are not present in the original tape, and OPS will not save any intermediate state for them. Instead, in the adjoint of the external function, the new tape will record one iteration and its intermediate states. However, OPS must reset the adjoint and primal states between each iteration, which involves runtime overheads.

3 ORCHESTRATION

As discussed in Section 2, all primal computational loops in OPS will write data at the center of the stencil or perform a reduction on global variables. OPS will save the state of the reductions for all loops in order to avoid recomputing reductions. For datasets, the parallel OPS loop already ensures that no race conditions exist for the writes; thus, creating copies of the states is performed inside the loops before calling the user kernel. In the adjoint loops, OPS will load these

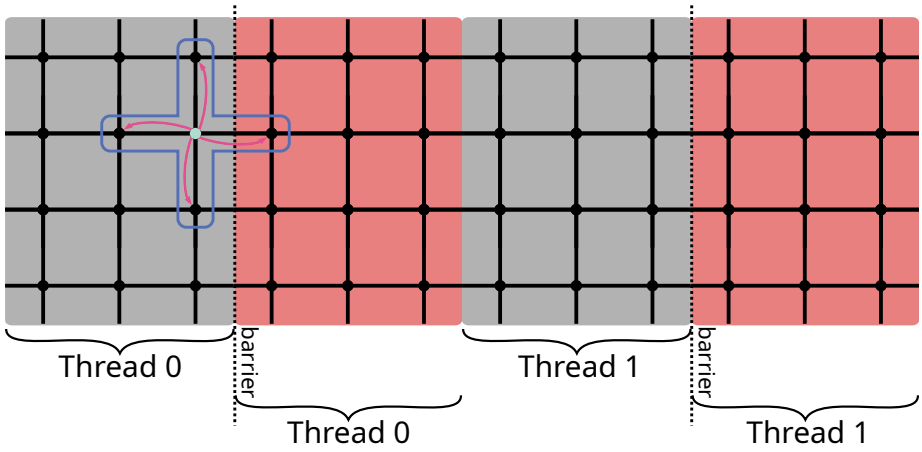
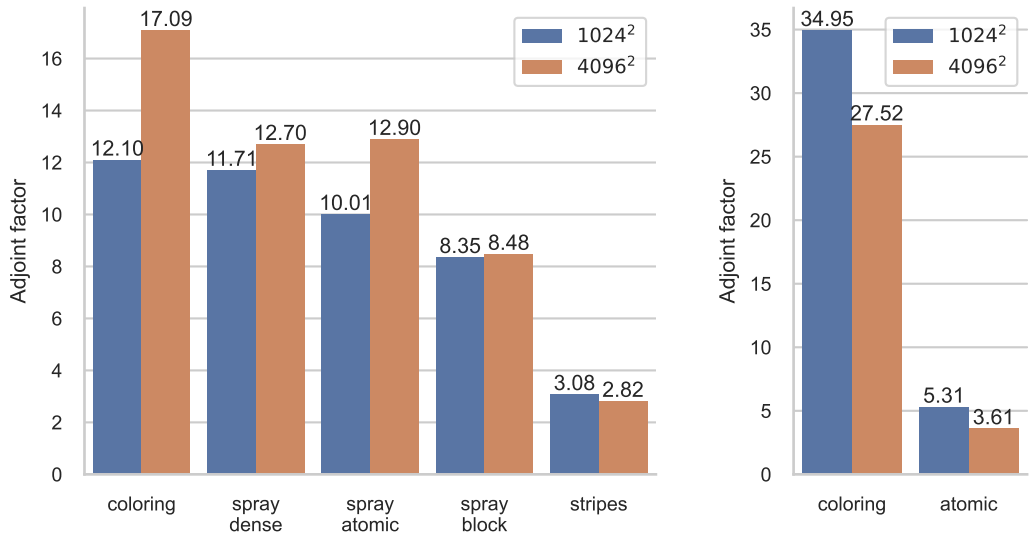


Fig. 3. Example scheduling applied by OPS with OpenMP for a three-point wide stencil. Stripes with the same color are executed parallel, each thread executes a single chunk.

copies for each grid point at the beginning of the loop body and redirect any subsequent writes for these datasets to local storage.

The adjoint kernels will reverse the data flow, introducing data races on the adjoint data of read-only datasets. There are four main data access patterns to consider: reductions, global reads (`ops_scalar`), data access on grid points, and data access on lower dimensional data. The first two will switch sides in the reverse pass on the adjoint data, the adjoint of reductions will be read-only data, and the loop will perform reductions on the adjoint for the `ops_scalar` arguments. In the other two cases, the data races will arise along read stencils accessing multiple points or lower dimensional (but still grid-dependent) data. In our implementation, we chose to handle the data races from lower dimensional data separately, which we will discuss in Section 3.1.

Without using high-level transformations on the code to transform the adjoint stencils into race-free gather stencils, there are a handful of approaches to ensure correct results with data races present during the adjoints. It is possible to use stencil-specific coloring [Lülfesmann and Kawarabayashi 2014] and run only points with the same color in parallel. Another commonly used technique is to use atomic operations on increments for data that is not thread-local or to use sparse or local reductions on the datasets with indirect increments. The deciding factors between these solutions is the cost of creating a coloring and running the loops in multiple chunks, the overhead of atomic operations, and the effect of the execution order on the locality of the processed points and on memory accesses. From the stencils in the primal loops, OPS has information on the pattern in which data races could arise during the adjoint of loops and can generate adjoint loops in different configurations. To compare these approaches on the adjoint loops we used a modified version of the adjoint of `hv predictor step 0` loop from the CDE application. This loop uses a twelve-point stencil to increment the adjoints of a 2D dataset and all the lower dimensional datasets are treated as passive data. As a baseline, we created a graph coloring with a greedy coloring algorithm that will not have any data races. On CPUs, we created three versions using the Spray [Hückelheim and Doerfert 2021] sparse reduction library. The first implementation (noted as *spray dense*) uses dense reductions similarly to the built-in reductions of OpenMP. These will allocate a thread local array for each thread, collect the increments, and then sum the increments from the threads at the end of the loop. The second version (*spray atomic*) uses atomic operations on every increment on the adjoints



(a) OpenMP (b) CUDA

Fig. 4. Adjoint factor of different scheduling approaches, on the adjoint of the simplified `hv_predictor_step_0` from the CDE application, with a mesh size of 1024^2 and 4096^2. The results using OpenMP in (a) are measured on a Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading using the Intel(R) oneAPI DPC++/C++ Compiler 2024.0.2, the CUDA measurements in (b) were performed on an NVidia A100. The adjoint factor is the relative runtime of the adjoint compared to the primal loop.

where the loop has data races. The final Spray implementation (*spray block*) uses the *BlockReduction* reducer which will divide the array into blocks of size 4096 and lazily allocate only the blocks where the thread actually writes data. These approaches have the advantage over coloring that they can keep the cache-friendly grid point processing order from the primal loops, but will have higher synchronization costs.

Finally, we introduce a two-color stripe coloring in OPS. Given the width of the stencils in a parallel loop, we divide the grid into stripes along the highest dimension. Figure 3 shows an example of the coloring. This coloring chooses the stripes, such that each stripe has a width equal to at least the width of the stencils. Then, the stripes are executed in two stages, and each stage computes at least the width of the stencil iterations, making both stages race-free (no overlaps of the written data within a stage).

Figure 4 shows the adjoint factors for these approaches. Each value shows the relative runtime of the adjoint loop compared to the primal loop. On CPUs (Figure 4a) the graph coloring approach leads to jumps between the processed grid points in each thread which will lead to poor memory access patterns and cache use. The sparse reduction approaches fix this issue, but at the cost of the overhead of reductions. The *stripes* approach takes the advantages of both worlds. Each thread will process one full stripe with nice memory access patterns and there is no extra synchronization overhead on indirect writes other than the synchronization between the two colors.

On GPUs, the overhead of introducing atomic operations is much smaller compared to the performance loss of the worse memory access patterns that would be introduced by colored execution. At the same time, introducing the *stripes* pattern on GPUs would add strided memory accesses which would also degrade performance. This explains the speedup

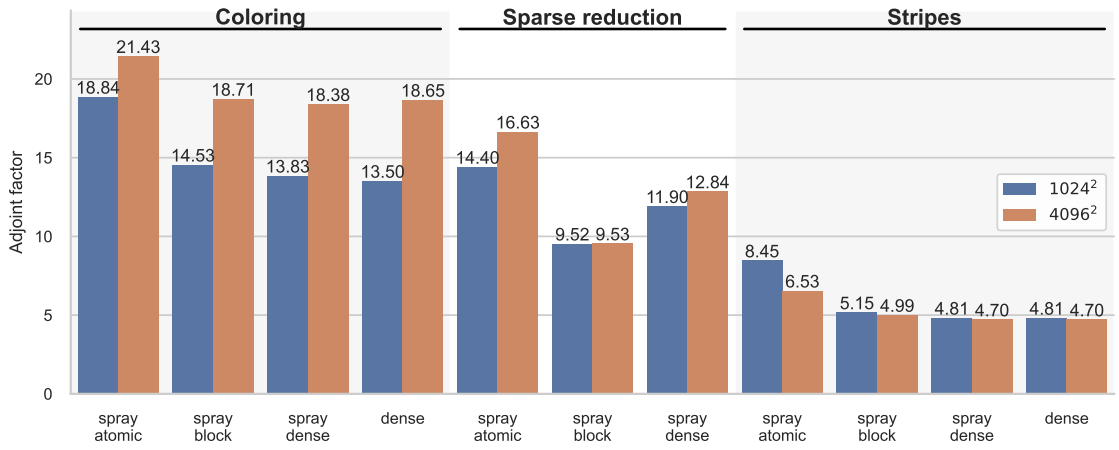


Fig. 5. Adjoint factor of different scheduling approaches on the adjoint of the `hv_predictor_step_0` from the CDE application, which has lower-dimensional data accesses, with a mesh size of 1024^2 and 4096^2. The measurements were made on a Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading using the Intel(R) oneAPI DPC++/C++ Compiler 2024.0.2. Labels on the top show the parallelization approach on the grid, while the X-axis labels show the handling of the increments on the adjoint of the lower-dimensional data.

of using atomics over graph coloring shown in Figure 4b. Therefore, adjoint loops in CUDA use atomic operations on the adjoints of datasets with a single kernel launch. This keeps the coalesced memory accesses in the loops, but the performance of the writes on the adjoint data depends on the stencil itself.

3.1 Handling low-dimension datasets

As mentioned previously, we handle access to lower dimensional datasets separately to mitigate the effect of the shared data access of large amounts of threads. Reads in the primal to a low-dimension dataset will result in a large number of writes in the adjoint loop to the adjoint of the low-dimension dataset. Essentially, the loops perform sums along the dimensions where the lower dimensional dataset is invariant. For example, in a two-dimensional loop such as in Listing 8, the loop would access two one-dimensional datasets x and v, where x is invariant along the Y axis, and v is invariant along the X axis.

```
for (int j = 1; j < size_y - 1; ++j) {
  for (int i = 1; i < size_x - 1; ++i) {
    u[j * size_x  + i] = x[i - 1] + x[i] + x[i + 1] + v[j - 1] + v[j] + v[j + 1];
  }
}
```

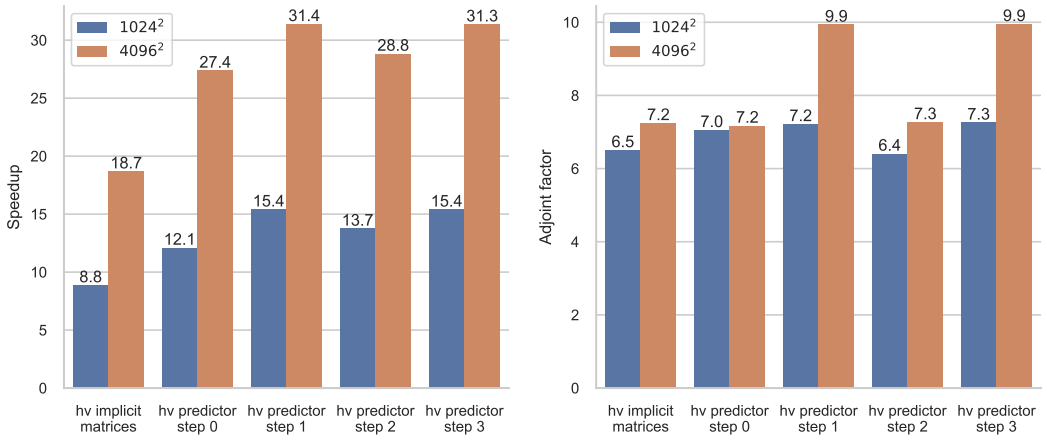
Listing 8. Example two-dimensional loop with data access to lower dimensional data.

In Figure 5 the shaded groups use the labeled approach at the top to handle the data races on the adjoints of normal datasets, and use the approach labeled on the X-axis for the lower-dimensional datasets. The *Coloring* labeled group uses general graph coloring on the normal datasets, *Sparse reduction* group uses the Spray [Hückelheim and Doerfert 2021] sparse reduction library with atomics, finally the *Stripes* group use the stencils on the normal datasets to create

the striped coloring. We see a similar picture as in Figure 4, with the notable exception that *spray dense* outperforms the atomic operations, due to the increased number of writes on the same location for the lower-dimensional arrays. Moreover, in the *stripes* implementations, the *spray dense* will become the most performant version of sparse reduction. Here, the loop is parallelized in the outermost (non-contiguous) dimension. If a dataset is not invariant in this dimension, such as v in the example, then the original parallelization already takes care of the races on the dataset. However, in the case of x, where the dataset is invariant in the outermost dimension, all threads would write the same values of the adjoint, which would lead to further data races. In these cases the threads will write all elements of these arrays, requiring *spray block* to also allocate the whole array in chunks (while losing performance in the process). The *dense* approach (which is used in OPS) is a slight modification of the *spray dense* approach with memory pooling to reduce the effect of always reallocating the thread private copies of the datasets.

On the GPU, atomic operations are sufficient for handling the data races in theory. However, in the presence of lower dimensional datasets, the performance of the atomic operations drastically drops due to the high number of conflicts. If possible, OPS will use two-dimensional CUDA thread blocks with the shape of $32 \times n_{warps}$ to reduce the effect of writes to the same location within an OPS block. This shape keeps the coalesced memory accesses of the primal on the higher dimensional datasets. However, threads within a warp will write the same values for $v[j]$ and $v[j \pm 1]$ from Listing 8 (a dataset that is invariant in X), so when the warp executes an atomic write, the memory address collisions within the warp will result in the write being serialized. In the case of x (a dataset that is invariant in Y), threads within a warp will only have address collisions if non-zero stencils are present, but address collisions arise among the warps inside the block. To reduce the effects of these data races, OPS will accumulate the increments of each thread in registers and then handle races in a separate step at the end of the kernel. There are four important cases that OPS handles. First, for datasets such as v that are always shared by the warp, OPS will perform a warp level reduction so that only one thread per warp will perform an atomic write to the global memory. Second, for datasets such as x that are always shared among the warps of the block, the increments are written with atomics directly by each thread at the end since the threads in a single warp do not have races in the center of the stencil. The final two cases arise in three- or higher-dimensional loops. If a dataset is invariant in both the X and Y dimensions, all threads within a block will share all accessed data, and OPS will perform a block-level reduction for all written adjoint values. Finally, datasets that are invariant only in the third or higher dimensions will behave like ordinary datasets from the perspective of the block.

Figure 6a shows the actual speedup this optimization achieved on the adjoints of kernels with lower-dimensional datasets from the CDE test application over using purely atomic operations on normal and lower-dimensional adjoints, and Figure 6b shows the achieved adjoint factors of the optimized adjoint loops. These kernels use the access pattern shown in Listing 8 on two separate datasets. The CDE application is a 2D stencil code on a non-uniform mesh [Wyns and Du Toit 2017]. Computing the finite difference estimates requires knowing the mesh coordinates to the left and right in the X dimension and to the top and bottom in the Y dimension, in addition to knowing the current mesh point. These X and Y mesh coordinates are stored in two 1D datasets. Warps have coalesced reads to the X mesh coordinates similarly to x in Listing 8, whereas each read from the Y mesh coordinates results in a single value broadcast across the warp as for v. The three-point access to the two datasets can be replaced with access to 12 separate datasets per mesh point but with a one-point stencil instead of three, which reduces the data races during the adjoint kernel. This optimization increases the performance for both the primal and adjoint kernels. In the presence of one-point stencils only, the reductions achieve 8.8 – 15.4x and 18.7 – 31.3x speedups on the 1024^2 and 4096^2 mesh, respectively. This speedup enables running the adjoint loops under 10x of the original primal loop on both problem sizes.



(a) Speedup of using the access-pattern-aware reductions on lower-dimensional datasets over atomics

(b) The runtime of the adjoint loops using the access-pattern-aware adjoint reductions relative to the primal

Fig. 6. Performance of our access-pattern-aware adjoint reductions and accumulations on lower-dimensional datasets on kernels from the CDE application with a mesh size of 1024^2 and 4096^2 on an NVidia A100. (a) Speedup over using only atomics, (b) Adjoint factor

4 EVALUATION

To show the performance of adjoint-mode AD in OPS, we test it on three distinct applications. We tested a **Poisson** mini application from the examples in OPS, the **Cloverleaf** [Mallinson et al. 2013] CFD application, and a 2D parabolic convection-diffusion reaction equation (**CDE**) solver from computational finance [Wyns and Du Toit 2017]. In our measurements, *passive* denotes the time it takes to run the original application, *forward* denotes the time it takes to run the application and record the tape, and *adjoint* denotes the time of computing derivatives for all inputs.

The Poisson application solves the equation

$$\nabla^2 u = f$$

with Jacobi iterations with 2D finite differences, where the update iteration can be written as:

$$u_{i,j}^{(n+1)} = \frac{(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)})dy^2 + (u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)})dx^2 - dx^2 dy^2 f_{i,j}}{2(dx^2 + dy^2)}$$

We used a 4096^2 mesh with 100 iterations for evaluating performance. These measurements show the performance of recording the tape and accumulating derivative information with OPS. We introduced two versions for handling the derivative propagation; the first is to record all iterations and compute the adjoints, and the second is to take advantage of the fact that the code can be expressed as a fixed point iteration. Hence, we can use reverse accumulation [Christianson 1994] and compute the derivative with a fixed point iteration with the tape for a single iteration.

CloverLeaf [Mallinson et al. 2013] is a mini-app that solves the compressible Euler equations on a Cartesian grid using an explicit, second-order accurate method. Cloverleaf is a mini-app from the UK Mini App Consortium and is widely used for performance benchmarks for stencil computations. We used the OPS implementation for Cloverleaf on two example problems, one with a 960^2 problem size and 2955 time steps and a bigger 3840^2 problem with 87 time steps.

The CDE application is a convection-diffusion-reaction code from computational finance [Wyns and Du Toit 2017]. It computes the price of a European call option driven by a Heston stochastic volatility model using the Method of

Intel(R) Xeon(R) Gold 6226R 16 threads using OpenMP	Revolve CPs	Runtime Passive (s)	Peak memory passive (GB)	Runtime Adjoint (s)	Peak memory adjoint (GB)	Adjoint factor	Number of sensitivities
Poisson 4096^2, 100 iter	-	0.677	0.54	2.974	15.98	4.39	34 M
Poisson Rev.Acc. 4096^2, 100 iter	-	0.677	0.54	2.514	4.97	3.72	34 M
Cloverleaf 960^2, 2955 iter	985	64.076	0.19	450.526	31.15	7.03	4 M
Cloverleaf 960^2, 2955 iter	100	64.076	0.19	469.480	4.25	7.33	4 M
Cloverleaf 3840^2, 87 iter	29	32.791	2.98	202.890	30.56	6.19	59 M
Cloverleaf 3840^2, 87 iter	10	32.791	2.98	218.522	19.79	6.66	59 M
CDE 1024^2, 100 iter	10	1.399	0.19	13.521	1.15	9.67	1 M
CDE 4096^2, 100 iter	10	34.259	3.09	261.440	13.44	7.63	17 M
NVidia A100							
Poisson 4096^2, 100 iter	-	0.032	0.54	0.195	15.98	6.05	34 M
Poisson Rev.Acc. 4096^2, 100 iter	-	0.032	0.54	0.171	4.97	5.31	34 M
Cloverleaf 960^2, 2955 iter	985	7.639	0.19	40.631	31.15	5.32	4 M
Cloverleaf 960^2, 2955 iter	100	7.639	0.19	42.825	4.25	5.61	4 M
Cloverleaf 3840^2, 87 iter	29	1.799	2.98	11.885	30.56	6.61	59 M
Cloverleaf 3840^2, 87 iter	10	1.799	2.98	12.722	19.79	7.07	59 M
CDE 1024^2, 100 iter	10	0.453	0.19	2.078	1.15	4.59	1 M
CDE 4096^2, 100 iter	10	3.731	3.09	20.157	13.44	5.40	17 M

Table 1. Runtime and memory comparison between the original applications and the computing of the adjoints measured on an Intel(R) Xeon(R) Gold 6226R with 16 threads using OpenMP (Top) and on an NVidia A100 using CUDA (Bottom). The values for Adjoint factor denote the runtime of the adjoint codes relative to the passive applications. **Note:** For the Poisson code, Rev.Acc. stands for using reverse accumulation, computing the adjoints as a fixed point iteration with storing tape for the last iteration only.

Lines. The CDE application uses a finite volume scheme with central finite differences and first-order upwinding at the $v = 0$ boundary. It uses the Hundsdorfer–Verwer ADI time stepping scheme and requires the solution of batches of tridiagonal systems, which the tridsolver [Balogh et al. 2022] library performs. We use this application to highlight two aspects: the effect of low-dimensional datasets on performance and external adjoint nodes in OPS.

4.1 Measurement Setup

We ran the OpenMP measurements on CentOS Linux release 7.9 using a single socket of an Intel(R) Xeon(R) Gold 6226R CPU at 2.9 GHz without hyper-threading and 376 GB RAM. The CUDA measurements were executed on a CentOS 8 using an AMD EPYC 7F72 24-core Processor and an NVidia A100 GPU with 40 GB RAM. All codes are compiled using GCC 11.2 and CUDA 12.0. We do not use Revolve for the Poisson application but save all intermediate states for the 100 iterations. When we do not specify otherwise, we report runtimes of Cloverleaf using 29 and 985 Revolve checkpoints for the 960^2 and 3840^2 test-cases respectively, and 10 checkpoints for CDE. All measurements were repeated ten times, and the mean value is shown as the final result. For each application, we measured the original application and the AD version with the combined forward and reverse pass.

4.2 Results

As with many stencil applications, the Poisson and Cloverleaf applications are memory bandwidth-bounded codes, where the achieved bandwidth is a commonly used metric for performance on a given hardware. The CDE application uses lower dimensional datasets in most computational kernels, which leads to compute-bound loops for most of the loops in the application.

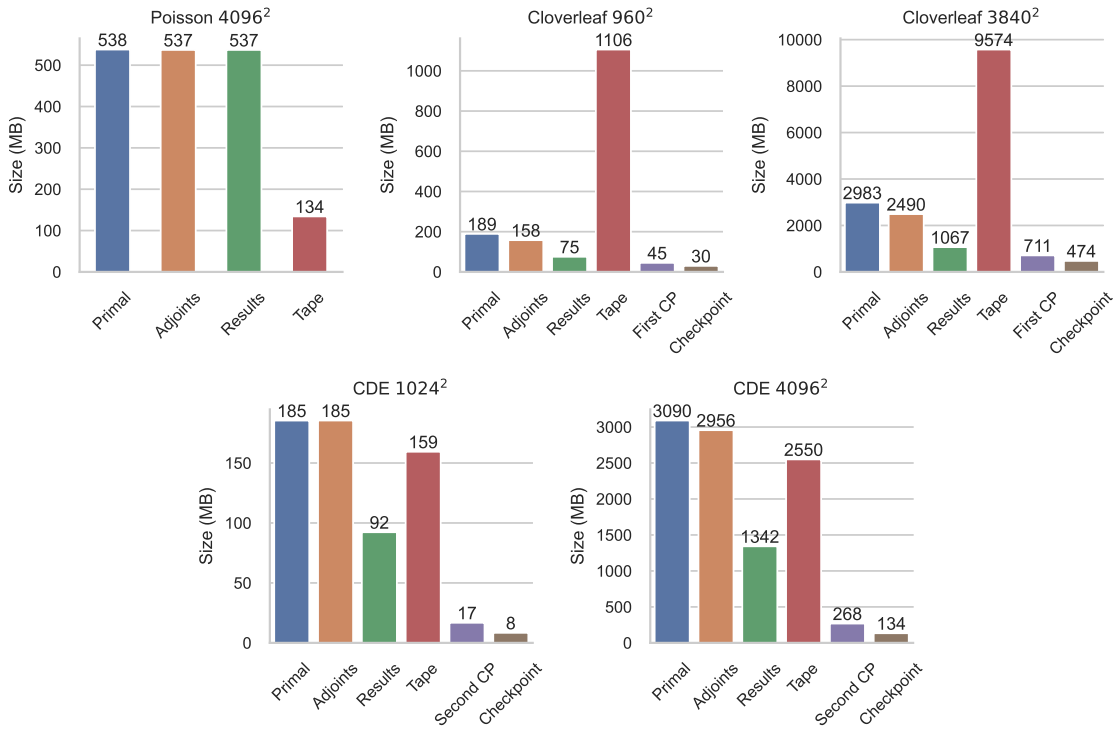


Fig. 7. Memory requirements for the applications. *Primal* is the memory usage of the original application, while *Adjoint* shows the adjoint memory allocated for datasets. The checkpoint sizes note the size of the Revolve checkpoints. *First CP* for Cloverleaf and *Second CP* for CDE marks a single checkpoint with more data due to datasets that are only written at the beginning. All other checkpoints use the uniform size shown as *Checkpoint*. *Tape* marks the required saved states for interpreting one time step. Finally, *Results* marks extra copies for saving the datasets at the end of the forward pass.

Table 1 shows all the absolute runtimes and peak memory usage of all the measured codes as well as the number of sensitivities computed by the evaluation of the adjoint model. The simplest application is Poisson, which performs the computations from Listing 4, where a single computational loop applies the stencil from Listing 1 in each iteration. The first version applies the adjoint calculation directly, caching all intermediate states of the datasets for the whole application during the forward pass. The *Rev.Acc.* version performs the 100 iterations without caching any state in an external adjoint function. It creates a new tape for the reverse accumulation during the adjoint evaluation, records a much smaller tape for a single iteration, and performs 100 adjoint evaluations on that tape. Figure 7 shows the components of the application's memory usage. The *Rev.Acc.* approach reduces the required memory footprint due to caching only one iteration (and a loop after the fixed-point iteration computing the error) and reusing that tape instead of using $100x$ of the memory storing the tape for 100 iterations. The trade-off is that this approach introduces overheads to the adjoint propagation corresponding to resetting the primal and adjoint memory between adjoint evaluation on the secondary tape. Nevertheless, these overheads have a lower impact on the total runtime than caching all intermediate states during the forward pass in our case.

(a) Bandwidth values measured on a single socket of an Intel(R) Xeon(R) Gold 6226R without hyper-threading using OpenMP.

Application	Testcase	Bandwidth (GiB/s)	Peak (%)
Poisson	4096^2, 100 iter	55.54	92.28
Poisson Active	4096^2, 100 iter	54.82	91.03
Poisson Rev.Acc.	4096^2, 100 iter	57.07	94.77
CloverLeaf	960^2, 2955 iter	50.33	83.58
CloverLeaf Active	960^2, 2955 iter	46.02	76.43
CloverLeaf	3840^2, 87 iter	59.42	98.67
CloverLeaf Active	3840^2, 87 iter	58.22	96.68
BabelStream Triad	Array size: 1.0 GiB	60.22	100

(b) Bandwidth values measured on the NVidia A100 GPU using CUDA.

Application	Testcase	Bandwidth (GiB/s)	Peak (%)
Poisson	4096^2, 100 iter	1195.93	92.87
Poisson Active	4096^2, 100 iter	980.11	76.11
Poisson Rev.Acc.	4096^2, 100 iter	968.66	75.22
CloverLeaf	960^2, 2955 iter	600.02	46.60
CloverLeaf Active	960^2, 2955 iter	634.70	49.29
CloverLeaf	3840^2, 87 iter	1124.07	89.13
CloverLeaf Active	3840^2, 87 iter	1024.86	79.59
BabelStream Triad	Array size: 1.0 GiB	1287.70	100

Table 2. Bandwidth values for the Poisson and Cloverleaf applications. The Peak (%) values show the relative bandwidth compared to the Triad kernel in BabelStream.

	OpenMP Throughput (GFLOPS/s)				CUDA Throughput (GFLOPS/s)			
	1024^2, 100 iter		4096^2, 100 iter		1024^2, 100 iter		4096^2, 100 iter	
	primal	adjoint	primal	adjoint	primal	adjoint	primal	adjoint
hv implicit matrices	40.90	41.14	41.29	47.13	1710	1390	1800	1340
hv predictor step 0	41.50	24.58	38.40	27.62	2850	1580	3000	1670
hv predictor step 1	36.95	36.90	37.01	41.92	2690	1950	3010	1800
hv predictor step 2	35.89	24.66	32.63	27.90	2720	1680	2900	1750
hv predictor step 3	36.99	35.95	37.01	41.31	2690	1950	3010	1800
Peak throughput	224.5				6280			

Table 3. Double precision throughput of compute bound kernels in the CDE application. The first four columns were measured on a single socket of an Intel(R) Xeon(R) Gold 6226R, the last four columns were measured on the NVidia A100 GPU, the peak throughputs measured with the Empirical Roofline Toolkit [Lo et al. 2015].

In Cloverleaf, we have significantly more kernels per time iteration (over 150), which leads to much more cached memory, as Figure 7 shows. Creating a tape containing data for every time step with these sizes is infeasible. Therefore, our implementation for Cloverleaf has to utilize Revolve.

Finally, in addition to Revolve, the CDE application uses the external function API with tridiagonal solver calls. We used two problem sizes to test a 1024^2 and a 4096^2 mesh with 100 time steps in each case. In both cases, the tridiagonal solver calls dominate the forward pass on GPUs (around 90% of total time), while on CPUs, the 1024^2 problem spends 23% and the 4096^2 46% of the forward pass solving tridiagonal systems.

We used the Triad kernel from BabelStream [Deakin et al. 2016] to measure the achievable bandwidth for our test platforms. This kernel achieved 60.22 GiB/s on a single NUMA node of an Intel(R) Xeon(R) Gold 6226R and 1287.70 GiB/s on the NVidia A100 GPU. Available memory bandwidth is the main performance bottleneck for the Poisson and Cloverleaf applications. Table 2a shows the achieved bandwidth for the whole application run on CPU. For all test cases, the adjoint versions achieve slightly lower bandwidth values compared to the primal due to the the additional synchronizations between the stripes, the biggest difference between the passive and the corresponding adjoint kernels

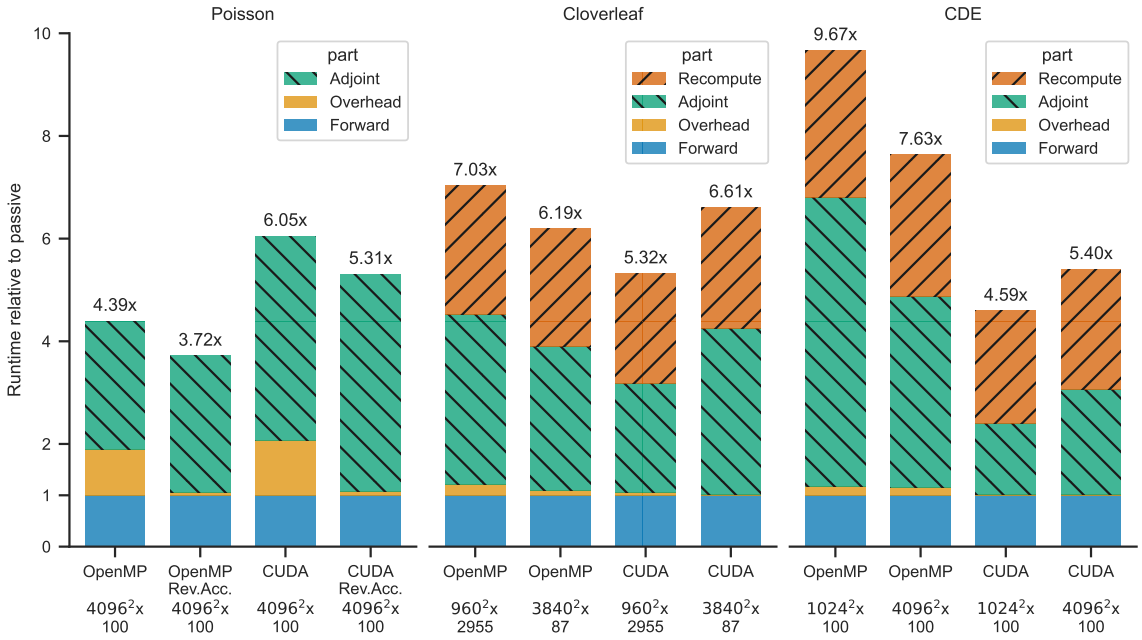


Fig. 8. The cost or adjoint factor of computing derivatives on the benchmark applications, relative to a single evaluation of the passive, original application, showed by **Forward**. All bars show the values relative to the original applications. The **Overhead** values show the additional cost of collecting the DAG, saving intermediate states, and storing Revolve checkpoints during the forward pass, **Recompute** values represent the additional time spent on replaying sections of the applications to restore states for the adjoints loops and the **Adjoint** part shows the time spent in the actual adjoint loops.

is 4.3 GiB/s for the 960^2 mesh in CloverLeaf which results in achieving 76.4% of the bandwidth of BabelStream. The highest bandwidth measurement for adjoints reaches 96.7% of BabelStream. Using reverse accumulation to compute the derivatives for the Poisson code achieves comparable bandwidth to the original application and lower runtime compared to computing the derivatives by recording all iterations due to lower overheads during the forward pass.

Table 2b shows similar results on the NVidia A100 GPU. The 960^2 mesh is too small to saturate the GPU and achieves worse bandwidth. Similarly to the OpenMP versions, the adjoint implementations achieve lower bandwidth in general, but in the case of CUDA, the impact of atomics leads to a bigger bandwidth drop compared to the coloring approach on CPUs.

Table 3 shows the achieved compute throughput for the compute-bound kernels from the CDE applications. The adjoint loops achieved similar compute throughput compared to the primal loops using OpenMP. However, on GPUs, the increased cost of the atomic leads to a larger difference in performance. As Table 3 shows, the adjoint loops achieve $55-82\%$ of the throughput of the corresponding primal loops while reaching up to 31% of the empirical double precision throughput of the A100 measured with the Empirical Roofline Toolkit [Lo et al. 2015] using the average arithmetic intensity of the adjoint kernels.

Note that the runtimes in Table 1 and the bandwidth values were measured using the same high-level OPS source code without any modifications.

Figure 8 shows the relation between the runtime of the original application and the adjoint evaluation. The total values at the end of each bar show the relative cost of calculating the adjoints (including evaluating the original code). The blue bars represent the original runtime, and the yellow *Overhead* parts represent the additional time the active code spends in the forward pass. This time is spent on tasks like caching intermediate states, building the tape, and handling Revolve checkpoints. The orange *Recompute* parts for Cloverleaf and CDE represent the time spent on re-running sections of the forward pass including storing and freeing data for Revolve checkpoints, and finally, the green part marks the time spent on actually evaluating the adjoints.

In the case of the Poisson code, we can clearly see the effect of caching intermediate states. For all loops for each grid point, OPS will save the value of the written dataset, doubling the write in each kernel, thus, the time required for the forward pass for OpenMP. The overhead is slightly bigger in the case of CUDA due to the higher cost of the memory operations related to the cached states. Nevertheless, the *Rev.Acc.* version clearly shows its advantage in reducing the overhead for the forward pass since it doesn't need to cache intermediate states during the fixed point iterations. Of course, the *Rev.Acc.* spends more time evaluating the adjoints as a consequence due to resetting the primal data and reseeding the adjoints between iterations of the reverse accumulation, increasing the cost of the adjoint evaluation from $2.5x$ to $2.7x$ and from $4.0x$ to $4.2x$ for OpenMP and CUDA respectively. These overheads are still compensated in the case of Poisson with the reduction in the *Overhead* cost, but this heavily depends on the application and the quality of the adjoint kernels as well.

The *Overhead* for Cloverleaf mainly comes from caching the final iteration and handling Revolve checkpoints. For each problem, we chose the maximum Revolve checkpoints as the third of the number of iterations, which shows the increased relative overhead for the 960^2 problem. However, the actual difference is amortized due to grouping the allocations for the checkpoints. The chosen checkpoint counts lead to spending $2.1 - 2.5x$ the time of the original application performing re-runs of forward kernels. In terms of actual adjoint kernels relative to the original application, OPS spends $3.3x$ (960^2) and $2.8x$ (4096^2) time on CPUs and $2.12x$ (960^2) and $3.2x$ (4096^2) time on GPUs.

For the CDE application, we fixed the checkpoint count to 10 for the 100 iteration, which leads to small overheads on the forward pass and $2.05 - 2.87x$ time spent on re-computing forward kernels (including forward kernels with caching). There is a large difference in the relative runtime of the adjoint computation between the OpenMP ($1024^2 - 5.62x, 4096^2 - 3.72x$) and the CUDA ($1024^2 - 1.4x, 4096^2 - 2.05x$) implementation. The main cause of the difference is in the time spent on the tridiagonal solver calls - the original CUDA application spends 92% for the small and 88% for the big mesh of the runtime in the tridiagonal solver calls, while for the OpenMP version this ratio is only 23% and 46% respectively. As we showed, the adjoint for the tridiagonal solve is another tridiagonal solve and a small loop on the derivatives leading to a small overhead on the adjoints. In terms of adjoint loop performance on the small mesh, both the OpenMP and CUDA versions compute the adjoints in $6.6 - 6.7x$ while we see a difference on the bigger mesh ($5.5x$ for OpenMP and $8.2x$ for CUDA) where the overhead from the reductions on the low dim datasets showing their effects with the increased number of blocks present.

Figure 8 clearly shows the effect of Revolve on the total runtime of the derivative propagation, and Figure 7 shows the effect on the memory consumption. The Poisson application shows the effect of naively implementing AD with caching all intermediate states. The original application uses 538 MB memory (4 datasets with a size of 134 MB each plus some additional memory in OPS). In addition, the active version allocates the adjoints for all these datasets and also saves an additional copy of the primal results. The value shown in Figure 7 for the tape corresponds to one time iteration where the loop writes one dataset. Hence, OPS will store 1 dataset worth of data for the time iteration, adding 134 MB per iteration, a total of 13.4 GB memory for the 100 iterations, and a runtime overhead on the forward pass,

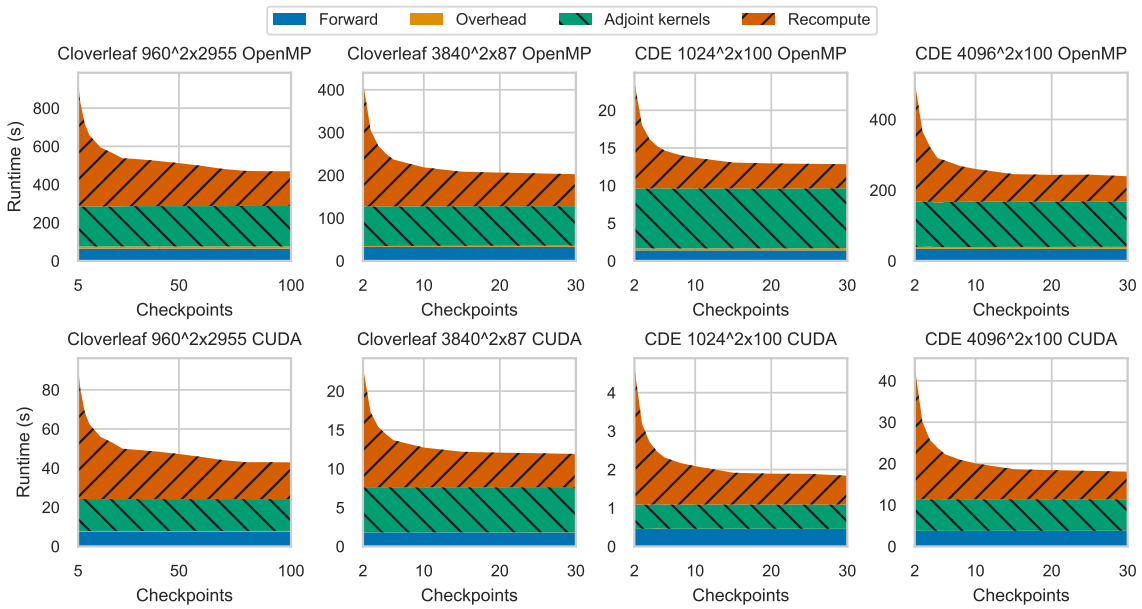


Fig. 9. Runtime of evaluating the adjoints for the applications (including the original applications) with Revolve. The number of available Revolve checkpoints is increased. The increased number of checkpoints reduces the number of additional forward steps required to restore intermediate states.

but potentially faster time to derivatives. This highlights the main benefit of reverse accumulation, since this version only caches data for one iteration, drastically decreasing the memory footprint of computing adjoints, resulting in a constant memory requirement for computing the adjoints instead of constantly increasing with the iteration count and the complexity of the computations. For a reasonably complex application, the size of the tape required for a single iteration drastically increases, making it infeasible to tape all iterations at once, and many applications are not fixed point iterations, ruling out reverse accumulation. For such applications, Revolve provides a solution to control the memory footprint at the cost of additional recomputing steps. Figure 7 also shows the memory cost of each Revolve checkpoints that the application stores data for, which determine how the memory requirements would change with the number of Revolve checkpoints used in Cloverleaf and CDE. The effect is twofold: OPS will cache only one iteration at a time (reducing the peak memory for this category by a factor of the number of time iterations and fixing it to the *Tape* value shown in the figure) and increasing the memory usage by saving the Revolve checkpoints to memory. Both applications have one-time writes on a few datasets, leading to a bigger initial checkpoint, but all other checkpoints are the same size, increasing the peak memory footprint linearly. Without the memory used for the Revolve checkpoints, OPS can compute the adjoints using only $5.4 - 8.1x$ more memory for Cloverleaf and $3.2 - 3.4x$ more memory for CDE compared to the original application. Counting the 10 Revolve checkpoints used for CDE OPS computes the adjoints under $3.7x$ more memory.

In terms of runtime, Revolve will, in total, increase the runtime with the additional re-compute steps and the cost of actually saving the checkpoints. To get a clearer picture of the effect on the runtime, Figure 9 shows both Cloverleaf and CDE runtimes at different maximum active checkpoint counts. The theoretical minimum runtime overhead would

arise if the application stores all Revolve checkpoints during the forward pass. Then, each iteration would be replayed once while saving all states. On the other end, using only a few Revolve checkpoints while having smaller memory requirements leads to significantly more re-compute steps. However, Figure 9 shows that, the overhead of the re-compute steps quickly drops as we increase the number of checkpoints, re-computing the forward steps only a handful of times, and allowing additional checkpoints to store to memory have diminishing returns. In addition, during re-computing loops, OPS can reduce the cost of execution by skipping reductions, whose results are already saved during the first execution and would significantly impact loop performance.

5 CONCLUSION

In this work, we presented an extension to the OPS DSL to support adjoint-mode algorithmic differentiation. OPS allows the expression of structured mesh applications from a high-level source and handles parallelism, data movement, and optimizations to different hardware backends. The extension provides performance portable derivatives for such applications on CPUs and GPUs. Our AD extension leverages the domain-specific abstraction of OPS to perform optimizations that would be more difficult with a more generic tool.

We used the description of the computational loops in the OPS abstraction to handle data dependencies and data races in the adjoint loops. We used this model to map the adjoint loops to OpenMP and CUDA implementations. Our approach builds a compact AD tape in OPS by handling derivatives at the loop level instead of expressions. Extensions to the tape, like support for external adjoint functions and optimal checkpointing via Revolve, were straightforward to implement using this high-level representation.

Experimental results on two memory bandwidth bound applications show that the adjoint propagation achieves similar bandwidth as the original primal application. On CPUs the applications achieved over 90% of the peak bandwidth on large grids, and only dropping 7% compared to the original on smaller test cases. Similarly, on GPUs, active versions achieve $75 - 80\%$ of the peak bandwidth on test cases that can saturate the GPU. Compared to the original applications, the OpenMP versions on all three test applications computed adjoints under $10x$ overhead and on GPUs under $4.6 - 6.6x$. We have integrated the Revolve algorithm into OPS and showed that we can control the memory requirements of the adjoint evaluation as expected. This will enable large simulations to run in adjoint-mode, which would require too much memory otherwise by trading memory requirements for runtime with re-computing sections of the forward pass.

Our adjoint model in OPS provides an efficient way to compute derivatives for performance-critical structured mesh applications. Using a DSL such as OPS unlocks the derivative computations on both hardware from the same high-level source, giving users access to a diverse selection of hardware with very different capabilities without any additional development costs.

ACKNOWLEDGMENTS

Gábor Dániel Balogh was supported by the ÚNKP-22-4 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. This research was supported by National Research, Development and Innovation Fund of Hungary (FK 145931), under the FK 23 funding scheme.

REFERENCES

Tim A. Albring, Max Sagebaum, and Nicolas R. Gauger. 2016. *Efficient Aerodynamic Design using the Discrete Adjoint Method in SU2.* American Institute of Aeronautics and Astronautics, Washington, D.C., 3518. https://doi.org/10.2514/6.2016-3518

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *2019 IEEE/ACM International Symposium on*

Code Generation and Optimization (CGO). 193–205. https://doi.org/10.1109/CGO.2019.8661197

Gabor D Balogh, Tobias S Flynn, Sylvain Laizet, Gihan R Mudalige, and István Z Reguly. 2022. Scalable many-core algorithms for tridiagonal solvers. *Computing in Science & Engineering* 24, 1 (2022), 26–35. https://doi.org/10.1109/MCSE.2021.3130544

Gábor Dániel Balogh and István Z. Reguly. 2020. Automatic parallel implementations of adjoint codes for structured mesh applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 908–911. https://doi.org/10.1109/CCGrid49817.2020.00019

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18 (2018), 1–43. http://jmlr.org/papers/v18/17-468.html

Kai Becker, Kathrin Heitkamp, and Edmund Kügeler. 2010. Recent progress in a hybrid-grid CFD solver for turbomachinery flows. In *Proceedings fifth European conference on computational fluid dynamics ECCOMAS CFD*, J. C. F. Pereira, A. Sequeira, and J. M. C. Pereira (Eds.), Vol. 2010. https://elib.dlr.de/68938/

David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

C.H. Bischof, H.M. Bucker, B. Lang, A. Rasch, and A. Vehreschild. 2002. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. 65–72. https://doi.org/10.1109/SCAM.2002.1134106

Christian Bischof, George Corliss, Larry Green, Andreas Griewank, Kara Haigler, and Perry Newman. 1992. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Computing Systems in Engineering* 3, 6 (1992), 625–637. https://doi.org/10.1016/0956-0521(92)90014-A

Johannes Blühdorn, Nicolas R Gauger, and Matthias Kabel. 2022. AutoMat: automatic differentiation for generalized standard materials on GPUs. *Computational Mechanics* 69, 2 (2022), 589–613. https://doi.org/10.1007/s00466-021-02105-2

Johannes Blühdorn, Max Sagebaum, and Nicolas Gauger. 2023. Event-Based Automatic Differentiation of OpenMP with OpDiLib. *ACM Trans. Math. Softw.* 49, 1, Article 3 (mar 2023), 31 pages. https://doi.org/10.1145/3570159

H. Martin Bücker, Bruno Lang, Dieter an Mey, and Christian H. Bischof. 2001. Bringing Together Automatic Differentiation and OpenMP. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy) *(ICS '01)*. Association for Computing Machinery, New York, NY, USA, 246–251. https://doi.org/10.1145/377792.377842

H. Martin Bücker, Arno Rasch, and Andreas Wolf. 2004. A Class of OpenMP Applications Involving Nested Parallelism. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (Nicosia, Cyprus) *(SAC '04)*. Association for Computing Machinery, New York, NY, USA, 220–224. https://doi.org/10.1145/967900.967948

Martin Bücker, George Corliss, Uwe Naumann, Paul Hovland, and Boyana Norris. 2006. *Automatic differentiation: applications, theory, and implementations*. Springer. https://doi.org/10.1007/3-540-28438-9

H.M. Bücker, B. Lang, A. Rasch, C.H. Bischof, and D. an Mey. 2002. Explicit loop scheduling in OpenMP for parallel automatic differentiation. In *Proceedings 16th Annual International Symposium on High Performance Computing Systems and Applications*. IEEE, 121–126. https://doi.org/10.1109/HPCSA.2002.1019144

Luca Capriotti and Jacky Lee. 2018. Case Studies of Real-Time Risk Management via Adjoint Algorithmic Differentiation (AAD). In *High-Performance Computing in Finance*. Chapman and Hall/CRC, 339–370. https://doi.org/10.1201/9781315372006-11

Alan Carle, Lawrence L Green, PA Newman, and CH Bischof. 1994. Applications of automatic differentiation in CFD. *NASA STI/Recon Technical Report N* 95 (1994), 16828.

Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. 2015. The Stan math library: Reverse-mode automatic differentiation in C++. *arXiv preprint arXiv:1509.07164* (2015). https://doi.org/10.48550/arXiv.1509.07164

Bruce Christianson. 1994. Reverse accumulation and attractive fixed points. *Optimization Methods and Software* 3, 4 (1994), 311–326. https://doi.org/10.1080/10556789408805572

A Clevenhaus, M Ehrhardt, and M Gunther. 2021. An ADI Sparse Grid method for Pricing Efficiently American Options under the Heston Model. *ADVANCES IN APPLIED MATHEMATICS AND MECHANICS* 13, 6 (2021), 1384–1397. https://doi.org/10.4208/aamm.OA-2020-0317

Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. GPU-STREAM v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *International Conference on High Performance Computing*. Springer, 489–507. https://doi.org/10.1007/978-3-319-46079-6_34

Bertram Düring, Michel Fournié, and Christof Heuer. 2014. High-order compact finite difference schemes for option pricing in stochastic volatility models on non-uniform grids. *J. Comput. Appl. Math.* 271 (2014), 247–266. https://doi.org/10.1016/j.cam.2014.04.016

H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

David M. Gay. 2006. Semiautomatic Differentiation for Efficient Gradient Computations. In *Automatic Differentiation: Applications, Theory, and Implementations*, Martin Bücker, George Corliss, Uwe Naumann, Paul Hovland, and Boyana Norris (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–158. https://doi.org/10.1007/3-540-28438-9_13

Ralf Giering. 2026. Direct Differentiation of Non-blocking MPI Library Calls. In *Proceedings of the 2024 International Conference on Algorithmic Differentiation (AD)*, H. Martin Bücker, Laurent Hascoët, Paul Hovland, Jan Hückelheim, Sri Hari Krishna Narayanan, Uwe Naumann, and Andrea Walther (Eds.).

SIAM, Philadelphia, PA, USA, 166–177. https://doi.org/10.1137/1.9781611979039.12

R. Giering, T. Kaminski, and T. Slawig. 2005. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems* 21, 8 (2005), 1345–1355. https://doi.org/10.1016/j.future.2004.11.003

Ralf Giering, Thomas Kaminski, Ricardo Todling, Ronald Errico, Ronald Gelaro, and Nathan Winslow. 2006. Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model. In *Automatic Differentiation: Applications, Theory, and Implementations*, Martin Bücker, George Corliss, Uwe Naumann, Paul Hovland, and Boyana Norris (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 275–284. https://doi.org/10.1007/3-540-28438-9_24

Michael B Giles and Niles A Pierce. 2000. An introduction to the adjoint approach to design. *Flow, turbulence and combustion* 65 (2000), 393–415. https://doi.org/10.1023/A:1011430410075

Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. 2008. Automatic Differentiation for GPU-Accelerated 2D/3D Registration. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 259–269. https://doi.org/10.1007/978-3-540-68942-3_23

Felix Gremse, Andreas Höfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. 2016. GPU-accelerated adjoint algorithmic differentiation. *Computer Physics Communications* 200 (2016), 300–311. https://doi.org/10.1016/j.cpc.2015.10.027

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45. https://doi.org/10.1145/347837.347846

Lluís Guasch, Oscar Calderón Agudo, Meng-Xing Tang, Parashkev Nachev, and Michael Warner. 2020. Full-waveform inversion imaging of the human brain. *NPJ digital medicine* 3, 1 (2020), 28. https://doi.org/10.1038/s41746-020-0240-8

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Transactions on Mathematical Software (TOMS)* 39, 3, Article 20 (may 2013), 43 pages. https://doi.org/10.1145/2450153.2450158

Patrick Heimbach, Chris Hill, and Ralf Giering. 2002. Automatic Generation of Efficient Adjoint Code for a Parallel Navier-Stokes Solver. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1019–1028. https://doi.org/10.1007/3-540-46080-2_107

Patrick Heimbach, Chris Hill, and Ralf Giering. 2005. An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation. *Future Generation Computer Systems* 21, 8 (2005), 1356–1371. https://doi.org/10.1016/j.future.2004.11.010

Robin J. Hogan. 2014. Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++. *ACM Trans. Math. Softw.* 40, 4, Article 26 (jul 2014), 16 pages. https://doi.org/10.1145/2560359

Cristian Homescu. 2011. Adjoints and automatic (algorithmic) differentiation in computational finance. *arXiv preprint arXiv:1107.1831* (2011). https://doi.org/10.48550/arXiv.1107.1831

Jan Hückelheim and Laurent Hascoët. 2022. Source-to-Source Automatic Differentiation of OpenMP Parallel Loops. *ACM Trans. Math. Softw.* 48, 1, Article 7 (feb 2022), 32 pages. https://doi.org/10.1145/3472796

Jan Hückelheim and Laurent Hascoët. 2023. Automatic Differentiation of Parallel Loops with Formal Methods. In *Proceedings of the 51st International Conference on Parallel Processing* (Bordeaux, France) *(ICPP '22)*. Association for Computing Machinery, New York, NY, USA, Article 13, 11 pages. https://doi.org/10.1145/3545008.3545089

Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. 2019. Automatic Differentiation for Adjoint Stencil Loops. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP '19)*. Association for Computing Machinery, New York, NY, USA, Article 83, 10 pages. https://doi.org/10.1145/3337821.3337906

J. C. Hückelheim, P. D. , Hovland, M. M. , Strout, , and J. D. Müller. 2018. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software* 33, 4-6 (11 2018), 672–693. https://doi.org/10.1080/10556788.2018.1435654

Willem H Hundsdorfer, Jan G Verwer, and WH Hundsdorfer. 2003. *Numerical solution of time-dependent advection-diffusion-reaction equations*. Springer Series in Computational Mathematics, Vol. 33. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-09017-6

Jan Hückelheim and Johannes Doerfert. 2021. Spray: Sparse Reductions of Arrays in OPENMP. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 475–484. https://doi.org/10.1109/IPDPS49936.2021.00056

Tim Kaler, Tao B. Schardl, Brian Xie, Charles E. Leiserson, Jie Chen, Aldo Pareja, and Georgios Kollias. 2021. *PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 144–158. https://doi.org/10.1137/1.9781611976489.11

Mahesh Lakshminarasimhan, Oscar Antepara, Tuowen Zhao, Benjamin Sepanski, Protonu Basu, Hans Johansen, Mary Hall, and Samuel Williams. 2024. Bricks: A high-performance portability layer for computations on block-structured grids. *The International Journal of High Performance Computing Applications* 38, 6 (2024), 549–567. https://doi.org/10.1177/10943420241268288

E. Larour, J. Utke, A. Bovin, M. Morlighem, and G. Perez. 2016. An approach to computing discrete adjoints for MPI-parallelized models applied to Ice Sheet System Model 4.11. *Geoscientific Model Development* 9, 11 (2016), 3907–3918. https://doi.org/10.5194/gmd-9-3907-2016

Soeren Laue. 2022. On the Equivalence of Automatic and Symbolic Differentiation. arXiv:1904.02990 [cs.SC] https://arxiv.org/abs/1904.02990

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (ToG)* 37, 4 (2018), 1–13. https://doi.org/10.1145/3197517.3201383

Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J. Ligocki, Matthew J. Cordery, Nicholas J. Wright, Mary W. Hall, and Leonid Oliker. 2015. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In *High Performance Computing Systems. Performance Modeling,*

Benchmarking, and Simulation, Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond (Eds.). Springer International Publishing, Cham, 129–148. https://doi.org/10.1007/978-3-319-17248-4_7

Johannes Lotz. 2016. *Hybrid approaches to adjoint code generation with dco/c++*. Dissertation. Department of Computer Science, RWTH Aachen University. http://publications.rwth-aachen.de/record/667318

Johannes Lotz, Uwe Naumann, Max Sagebaum, and Michel Schanen. 2013. Discrete Adjoints of PETSc through dco/c++ and Adjoint MPI. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 497–507. https://doi.org/10.1007/978-3-642-40047-6_51

Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A Witte, Felix J Herrmann, Paulius Velesko, and Gerard J Gorman. 2019. Devito (v3. 1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development* 12, 3 (2019), 1165–1187. https://doi.org/10.5194/gmd-12-1165-2019

Michael Lülfesmann and Ken-ichi Kawarabayashi. 2014. Sub-exponential graph coloring algorithm for stencil-based Jacobian computations. *Journal of Computational Science* 5, 1 (2014), 1–11. https://doi.org/10.1016/j.jocs.2013.06.002

Andrew Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. 2013. Cloverleaf: Preparing hydrodynamics codes for exascale. *The Cray User Group* 2013 (2013).

Charles C Margossian. 2019. A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 9, 4 (2019), e1305. https://doi.org/10.1002/widm.1305

Reza Mollapourasl, Majid Haghi, and Alfa Heryudono. 2020. Numerical simulation and applications of the convection–diffusion–reaction equation with the radial basis function in a finite-difference mode. *Journal of Computational Finance* 23, 5 (2020). https://doi.org/10.21314/JCF.2020.382

William Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12472–12485. https://proceedings.neurips.cc/paper_files/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf

William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

William S Moses, Sri Hari Krishna Narayanan, Ludger Paehler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. 2022. Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–18. https://doi.org/10.1109/SC41404.2022.00065

Gihan R Mudalige, IZ Reguly, Satya P Jammy, Christian T Jacobs, Michael B Giles, and Neil D Sandham. 2019. Large-scale performance of a DSL-based multi-block structured-mesh application for Direct Numerical Simulation. *J. Parallel and Distrib. Comput.* 131 (2019), 130–146.

Jens-Dominik Müller, Jan Hückelheim, and Orest Mykhaskiv. 2018. *STAMPS: a Finite-Volume Solver Framework for Adjoint Codes Derived with Source-Transformation AD*. 2928. https://doi.org/10.2514/6.2018-2928

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. https://doi.org/10.1016/j.procs.2010.04.206 ICCS 2010.

Uwe Naumann. 2011. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. SIAM. https://doi.org/10.1137/1.9781611972078

Eric Phipps, Roger Pawlowski, and Christian Trott. 2022. Automatic Differentiation of C++ Codes on Emerging Manycore Architectures with Sacado. *ACM Trans. Math. Softw.* 48, 4, Article 43 (dec 2022), 29 pages. https://doi.org/10.1145/3560262

Eric T. Phipps, Roscoe A. Bartlett, David M. Gay, and Robert J. Hoekstra. 2008. Large-Scale Transient Sensitivity Analysis of a Radiation-Damaged Bipolar Junction Transistor via Automatic Differentiation. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–362. https://doi.org/10.1007/978-3-540-68942-3_31

Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. 2011. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '11)*. Association for Computing Machinery, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/2016604.2016641

Eric Raut, Jie Meng, Mauricio Araya-Polo, and Barbara Chapman. 2020. Evaluating performance of OpenMP tasks in a seismic stencil application. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*. Springer, 67–81. https://doi.org/10.1007/978-3-030-58144-2_5

István Z. Reguly, Gihan R. Mudalige, and Michael B. Giles. 2018. Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 873–886. https://doi.org/10.1109/TPDS.2017.2778161

Istvan Z. Reguly, Andrew M. B. Owenson, Archie Powell, Stephen A. Jarvis, and Gihan R. Mudalige. 2021. Under the Hood of SYCL – An Initial Performance Analysis with An Unstructured-Mesh CFD Application. In *High Performance Computing*, Bradford L. Chamberlain, Ana-Lucia Varbanescu, Hatem Ltaief, and Piotr Luszczek (Eds.). Springer International Publishing, Cham, 391–410. https://doi.org/10.1007/978-3-030-78713-4_21

Ruyman Reyes and Victor Lomüller. 2016. SYCL: Single-source C++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 673–682. https://doi.org/10.3233/978-1-61499-621-7-673

M. Sagebaum, T. Albring, and N. R. Gauger. 2018. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optimization Methods and Software* 33, 4-6 (2018), 1207–1231. https://doi.org/10.1080/10556788.2018.1471140

Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2019. High-Performance Derivative Computations Using CoDiPack. *ACM Trans. Math. Softw.* 45, 4, Article 38 (dec 2019), 26 pages. https://doi.org/10.1145/3356900

R Sanchez, T Albring, R Palacios, NR Gauger, TD Economon, and JJ Alonso. 2018. Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation. *Internat. J. Numer. Methods Engrg.* 113, 7 (2018), 1081–1107. https://doi.org/10.1002/nme.5700

Michel Schanen, Michael Förster, Johannes Lotz, Klaus Leppkes, and Uwe Naumann. 2012. Adjoining hybrid parallel code. In *Proceedings of the eighth international conference on engineering computational technology*, Vol. 100. Citeseer, Dubrovnik, Croatia, 18. https://doi.org/10.4203/ccp.100.7

Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. 2010. Interpretative adjoints for numerical simulation codes using MPI. *Procedia Computer Science* 1, 1 (2010), 1825–1833. https://doi.org/10.1016/j.procs.2010.04.204 ICCS 2010.

Gagandeep Singh, Alireza Khodamoradi, Kristof Denolf, Jack Lo, Juan Gomez-Luna, Joseph Melber, Andra Bisca, Henk Corporaal, and Onur Mutlu. 2023. SPARTA: Spatial Acceleration for Efficient and Scalable Horizontal Diffusion Weather Stencil Computation. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 463–476. https://doi.org/10.1145/3577193.3593719

Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. *SIGPLAN Not.* 49, 6 (June 2014), 65–76. https://doi.org/10.1145/2666356.2594342

M. Towara and U. Naumann. 2013. A Discrete Adjoint Model for OpenFOAM. *Procedia Computer Science* 18 (2013), 429–438. https://doi.org/10.1016/j.procs.2013.05.206 2013 International Conference on Computational Science.

Markus Towara, Michel Schanen, and Uwe Naumann. 2015. MPI-Parallel Discrete Adjoint OpenFOAM. *Procedia Computer Science* 51 (2015), 19–28. https://doi.org/10.1016/j.procs.2015.05.181 International Conference On Computational Science, ICCS 2015.

Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. 2008. OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes. *ACM Trans. Math. Softw.* 34, 4, Article 18 (jul 2008), 36 pages. https://doi.org/10.1145/1377596.1377598

Tom Verstraete, Lasse Müller, and Jens-Dominik Müller. 2017. Adjoint-Based Design Optimisation of an Internal Cooling Channel U-Bend for Minimised Pressure Losses. *International Journal of Turbomachinery, Propulsion and Power* 2, 2 (2017), 10. https://doi.org/10.3390/ijtpp2020010

S. Vitale, M. Pini, and P. Colonna. 2020. Multistage Turbomachinery Design Using the Discrete Adjoint Method Within the Open-Source Software SU2. *Journal of Propulsion and Power* 36, 3 (2020), 465–478. https://doi.org/10.2514/1.B37685

Yin Wang, Kun Hua, and Jun Zhang. 2011. Fast and High Accuracy Numerical Methods for Solving PDEs in Computational Finance. In *2011 International Conference on Business Computing and Global Informatization*. 307–310. https://doi.org/10.1109/BCGIn.2011.84

Maarten Wyns and Jacques Du Toit. 2017. A finite volume–alternating direction implicit approach for the calibration of stochastic local volatility models. *International Journal of Computer Mathematics* 94, 11 (2017), 2239–2267. https://doi.org/10.1080/00207160.2017.1297805

Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. 30–39. https://doi.org/10.1109/WOLFHPC.2016.08

Emre Özkaya and Nicolas R. Gauger. 2010. Automatic transition from simulation to one-shot shape optimization with Navier-Stokes equations. *GAMM-Mitteilungen* 33, 2 (2010), 133–147. https://doi.org/10.1002/gamm.201010011

Received 20 February 2007