2023

Efficient GPU Implementation of Automatic Differentiation for Computational Fluid Dynamics

Mohammad Zubair
Old Dominion University

Desh Ranjan
Old Dominion University

Aaron Walden
NASA Langley Research Center

Gabriel Nastac
NASA Langley Research Center

Eric Nielsen
NASA Langley Research Center

See next page for additional authors

Original Publication Citation

Zubair, M., Ranjan, D., Walden, A., Nastac, G., Nielsen, E., Diskin, B., Paterno, M., Jung, S., & Davis, J. H. (2023). Efficient GPU implementation of automatic differentiation for computational fluid dynamics (Report No. FERMILAB-CONF-23-342-CSAID). Fermilab. https://inspirehep.net/literature/2679722

Efficient GPU Implementation of Automatic Differentiation for Computational Fluid Dynamics

Mohammad Zubair and Desh Ranjan
Old Dominion University
Norfolk, Virginia

Aaron Walden, Gabriel Nastac, and Eric Nielsen
NASA Langley Research Center
Hampton, Virginia

Boris Diskin
National Institute of Aerospace
Hampton, Virginia

Marc Paterno
Fermi National Accelerator Laboratory
Batavia, Illinois

Samuel Jung
Northwestern University
Evanston, Illinois

Joshua Hoke Davis
University of Maryland
College Park, Maryland

***Abstract*—Many scientific and engineering applications require repeated calculation of derivatives of output functions with respect to input parameters. Automatic Differentiation (AD) is a methodology that automates derivative calculation and can significantly speed up the code development. In Computational Fluid Dynamics (CFD), derivatives of flux functions with respect to state variables (Jacobian) are needed for efficient solution of nonlinear governing equations. AD of the flux function on graphics processing units (GPUs) is challenging as flux computation involves many intermediate variables that create a high register pressure and requires significant memory traffic because of the need to store the derivatives.**

This paper presents a forward-mode AD method based om multivariate dual numbers that addresses these challenges and simultaneously reduces the operation count. The dimension of multivarite dual numbers is optimized for performance. The flux computations are restructured to minimize the number of temporary variables and reduce the register pressure. For effective utilization of the memory bandwidth, we use shared memory to store the local Jacobian. The threads assigned to process an edge (dual-face) collectively populate the local Jacobian in the shared memory.

Shared memory is used to store local flux Jacobian. The threads assigned to process a flux differentiation at an edge collectively populate the local Jacobian in the shared memory. The use of shared memory allows further reducing temporary variables. The local Jacobian is written from the shared memory to the device memory taking advantage of coalesced stores. This is another major benefit of the shared memory approach. During this work, we assessed existing GPU-based forward-mode AD approaches for flux Jacobian computation and found them performing suboptimally. We demonstrated that our GPU implementation based on multivariate dual numbers of dimension 5 outperforms other tested implementations including the hand-differentiated version optimized for NVIDIA V100. Our implementation achieves 75% of the peak floating point throughput and 61% of the peak device bandwidth on V100.

We present solutions to address these challenges while taking advantage of reducing the operation count in forward mode AD simultaneously via the use of multivariate dual numbers. This includes the optimal choice of the dimension of the multivariate dual numbers to be used. Additionally, we restructure the computation to minimize the number of temporary variables. For effective utilization of the memory bandwidth, we use shared memory to store the local Jacobian. The threads assigned to process an edge (dual-face) collectively populate the local Jacobian in the shared memory. Next, we write the local Jacobian from the shared memory to the device memory. The use of shared memory helps reduce temporary arrays or variables, which can be very expensive. The other major benefit is to perform coalesced stores when writing from shared memory to the device memory. We created and analyzed the performance of different GPU implementations. We demonstrated that the GPU implementation based on multivariate dual numbers of size 5 (perfect compressible gas) outperforms all other implementations including the optimized hand-differentiated version on an NVIDIA V100. This implementation achieves 75% of the peak floating point throughput and 61% of the peak device bandwidth on V100.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Many scientific and engineering applications require repeated calculation of derivatives of output functions with respect to input parameters. In Computational Fluid Dynamics (CFD), derivatives of flux functions with respect to state variables (Jacobian) are needed for efficient solution of nonlinear governing equations. Optimization, error estimation, uncertainty quantification and other applications require derivatives of selected objective functions with respect to a specified set of input parameters that may be counted in thousands. Often, the differentiated functions are expressed via a coded algorithm. Apart from mathematical operations, the code for a mathematical function may contain loops, branches, and other programming constructs. Differentiation of such a code is a complex task that may require significant development efforts and computational resources. With increased interest in machine learning and quick deployment of large complex models, efficient implementation of derivatives on graphics-processing units (GPUs) becomes critical. This paper studies GPU implementations of Automatic Differentiation (AD) for a CFD application.

Traditionally, CFD derivatives are computed by hand differentiating the target functions and coding the resulting derivative formulations. Hand differentiation is a manual labor-intensive process that is prone to errors. Reference [1] illustrates the complexity of a hand-differentiated formulation for a practical application in aerodynamics. AD is an alternative

Identify applicable funding agency here. If none, delete this.

methodology that automates derivative calculation and can significantly speed up the code development.

The AD approaches can be classified under two categories: source code transformation and compile-time/run-time approaches. Source code transformation refers to creation of a separate derivative code from the source code. This is sometimes referred to as a "static approach" as the code for computing the derivative is created before the compilation process. In the compile-time/run-time approach, sometimes referenced as a "dynamic approach", no separate code is created. The derivatives of an output function are computed together with the function itself by declaring the source code variables as suitable "structures" and using overloaded operators.

Most real-valued mathematical functions used in scientific and engineering applications are compositions of simple functions. Hence the derivative of an output function can be computed by repeatedly using the chain rule. For a composite function $f(g(x), h(x))$, the derivative, $\frac{df}{dx}$, is computed as follows.

$$\frac{df}{dx} = \frac{\partial f}{\partial g} \times \frac{dg}{dx} + \frac{\partial f}{\partial h} \times \frac{dh}{dx}$$

Different AD methods can be interpreted as a chain rule executed either in "forward mode" or "reverse mode" [2]. In the forward mode, the chain rule is executed from right to left meaning that AD computes derivatives of all intermediate quantities with respect to a particular input parameter. In the reverse mode, the chain rule is executed from left to right meaning that AD computes all derivatives of a particular output function with respect to all intermediate quantities.

The forward mode is often easier to implement and has a smaller memory footprint. The complexity of an efficient forward-mode AD implementation is proportional to the number of input parameters. One forward-mode AD method that is widely used in CFD and multidisciplinary optimization is based on the complex-number theory [3]–[5]. Many forward-mode AD implementations use the dynamic approach, for example, [6]–[9]. The reverse-mode AD has the complexity that is proportional to the number of output functions and is suitable for applications where there are many input parameters and few output functions. The reverse mode is more challenging to implement and parallelize. Its implementation often requires additional memory or/and computations [10]–[14]. Depending on the underlying GPU architecture and the application, the registers required for the reverse-mode implementation can far exceed the available registers resulting in spills and significant performance slowdown. In [15], authors demonstrated that an efficient forward-mode AD implementation outperforms the most-efficient reverse-mode AD implementation for a function with several hundreds of input parameters. Note that in such a case, the forward mode actually performs significantly more floating point operations than the reverse mode; the superior performance is achieved due to reduction of the memory-related overhead.

Recently, several AD tools have been developed for GPU architectures, for example, [16]–[20]. Enzyme [16] performs reverse-mode AD of GPU kernels using a LLVM-based plugin [18] that can generate kernel gradients in CUDA or ROCm. In [16], the authors demonstrated that the AD performance on a set of benchmarks is within an order of magnitude of the performance of the original program. Sacado [17] implements forward-mode AD using operator overloading with expression templates of C++ programs. The GPU support for Sacado is accomplished through Kokkos [21], a C++ portable framework that works with GPUs from different vendors.

In this paper, we focus on Jacobian computations for discrete equations that arise in FUN3D [22], a CFD software developed at the NASA Langley for aerodynamics applications across the speed range. FUN3D plays a critical role in many national programs, including numerous science and engineering efforts across all mission directorates at NASA, other government agencies, major companies across the aerospace industry, and a large number of academic institutions.

To understand the challenges of Jacobian computations, we briefly describe the computations that are involved in evaluation of discrete residuals of the governing conservation-law equations implemented in FUN3D. The residuals of discrete equations are evaluated at grid points of an unstructured grid. The residuals are obtained by summing the flux contributions from grid edges which are dual-faces. This summation is accomplished in a loop over edges. For each edge, a function `fluxkernel` is called to compute the flux associated with the edge. The edge flux contributes to the residuals at two endpoints of the edge. The inputs to the `fluxkernel` function are $2n_b$ real numbers representing solution components. Here, n_b is the number of governing equations. The output of the `fluxkernel` function is a real vector of size n_b. The calculation of the flux at an edge is independent of flux calculations at all other edges. The residual computation is parallelized over edges. Care should be taken to avoid race conditions, which can occur when two threads operating on different edges try to write to the same location corresponding to a residual at a grid point.

The global Jacobian matrix represents the derivative of the residual vector with respect to the vector of solution components. The Jacobian assembly is also parallelized over edges. At each edge, the derivatives of the flux vector are computed with respect to the inputs of the `fluxkernel` function. The resulting local flux Jacobian matrices contribute to the global Jacobian matrix.

There are several challenges for developing a high-performance Jacobian implementation. The flux computation within the `fluxkernel` function involves many intermediate variables that create a high register pressure and requires n_b memory updates per edge. The Jacobian computations further increase the register pressure. The need to aggregate the local flux Jacobian into a global Jacobian matrix results in a significant increase in memory traffic. The derivative of the `fluxkernel` function locally generates $2n_b \times n_b$ size output that is used to build the global Jacobian matrix. In FUN3D, the diagonal part of the global Jacobian matrix (`Adiag`) is kept separately from the off-diagonal part (`Aoff`). `Adiag` is stored

with double precision, `Aoff` is stored with single precision. The two $n_b \times n_b$ local Jacobian blocks append two $n_b \times n_b$ blocks of the `Adiag` matrix and the two $n_b \times n_b$ blocks of the `Aoff` matrix.

This paper presents a forward-mode AD method that addresses these Jacobian GPU implementation challenges and simultaneously reduces the operation count via use of multivariate dual numbers. The dimension of multivarite dual numbers has been optimized for performance. The flux computations have been restructured to minimize the number of temporary variables and reduce the register pressure. Shared memory is used to store local flux Jacobian. The threads assigned to process a flux differentiation at an edge collectively populate the local Jacobian in the shared memory. The use of shared memory allows further reducing temporary variables. The local Jacobian is written from the shared memory to the device memory taking advantage of coalesced stores. This is another major benefit of the shared memory approach. During this work, we assessed existing GPU-based forward-mode AD approaches for flux Jacobian computation and found them performing suboptimally. We demonstrated that our GPU implementation based on multivariate dual numbers of dimension 5 outperforms other tested implementations including the hand-differentiated version optimized for NVIDIA V100. Our implementation achieves 75% of the peak floating point throughput and 61% of the peak device bandwidth on V100.

The paper is organized as follows. Section II gives background on AD. Section III details the various implementations investigated. Lastly, results are presented in Section IV.

II. Automatic Differentiation with Dual Numbers

In this section, we review AD implementations based on simple and multivariate dual numbers. Operation count and memory requirements associated with such implementations are analyzed.

A. Simple Dual Numbers

A simple dual number (or dual) is an ordered pair of real numbers. A dual-number forward-mode AD can be implemented as follows. All real numbers involved in function evaluation are replaced by duals. For each quantity u, the dual corresponding to u stores the value of u and the value of $\frac{du}{dx}$. The mathematical operations are extended to duals. For example, for duals $u = \langle a_0, b_0 \rangle$ and $v = \langle a_1, b_1 \rangle$, we can define the operations $+, \times, \texttt{sin}$, and $\texttt{sqrt}$ as below:

$$u + v = \langle a_0 + a_1, b_0 + b_1 \rangle$$
$$u \times v = \langle a_0 * a_1, a_0 * b_1 + b_0 * a_1 \rangle$$
$$\texttt{sin}(u) = \langle \texttt{sin}(a_0), \texttt{cos}(a_0) * b_0 \rangle$$
$$\texttt{sqrt}(u) = \langle \texttt{sqrt}(a_0), (0.5/\texttt{sqrt}(a_0)) * b_0 \rangle$$

Recursively performing the operations, one can carry forward both the real quantity in the first component and its derivative in the second component. The differentiated functions can be multivariate, i.e., $f = f(x_1, \ldots, x_n)$. The forward mode considers derivatives with respect to a single independent input parameter x_k. By convention, $\frac{dx_j}{dx_k} = 0$ if $j \neq k$ and $\frac{dx_k}{dx_k} = 1$. Hence, to compute the derivative $\frac{df}{dx_k}$ at $x_1 = r_1, \ldots, x_n = r_n$, we initialize the corresponding duals to $\langle r_1, 0 \rangle, \langle r_2, 0 \rangle, \ldots, \langle r_k, 1 \rangle, \ldots, \langle r_n, 0 \rangle$. To obtain the derivatives with respect to several or all input parameters, one can repeat the computation with each desired value of k. The only thing that changes is the initialization of the duals.

A major advantage of the dual-number AD implementation is that it requires only minor modifications of the source code. One can define a new type for duals with corresponding dual operators as a standalone library. Once this is done, one can change all real variables in the function calculation to duals. The operator overload mechanism provided in languages like C++ automatically takes care of applying the dual operators where appropriate.

B. Multivariate Dual Numbers

Calculation of multiple derivatives of a multivariate function using simple duals as described above incurs a penalty of repeated computations, for example, those performed for computing the function itself. One way to alleviate this penalty is to compute several derivatives simultaneously. This can be done by using multivariate dual numbers (or multivariate duals). A multivariate dual is a vector. The first component of the vector stores the quantity itself, the other components store the derivatives of the quantity with respect to specified input parameters. For example, if we want to compute the derivatives of f with respect to two independent parameters, x_k and x_l, we can define a multivariate dual with three components. The first component is f, the second component is $\frac{df}{dx_k}$, and the third component is $\frac{df}{dx_l}$. The corresponding dual operators can be defined as follows. If $u = \langle a_0, b_0, c_0 \rangle$ and $v = (a_1, b_1, c_1)$,

$$u + v = \langle a_0 + a_1, b_0 + b_1, c_0 + c_1 \rangle$$
$$u \times v = \langle a_0 * a_1, a_0 * b_1 + b_0 * a_1, a_0 * c_1 + c_0 * a_1 \rangle$$
$$\texttt{sin}(u) = \langle \texttt{sin}(a_0), \texttt{cos}(a_0) * b_0, \texttt{cos}(a_0) * c_0 \rangle$$
$$\texttt{sqrt}(u) =$$
$$\langle \texttt{sqrt}(a_0), (0.5/\texttt{sqrt}(a_0)) * b_0, (0.5/\texttt{sqrt}(a_0) * c_0 \rangle$$

GN: not checking for 0 or negative values for sqrt in snippet

Note that use of multivariate duals avoids re-computation not only in the first component but also across other components, e.g., for the derivative of `sin(u)`, $\texttt{cos}(a_0)$ can be calculated only once and reused.

Let's assume that we have a function $f = f(x_1, \ldots, x_n)$ of n independent input parameters and need to compute its derivatives with respect to all input parameters. We can do so by using simple duals and computing the function n times (once for each input parameter). In the simple-dual implementation, f is executed n times. If f contains a multiplication instruction $u*v$, the number of scalar operations needed to execute it over n iterations is $3n$ multiplications and n additions since each simple-dual multiplication takes three scalar multiplications and one scalar addition. In contrast, if we use multivariate dual numbers with $n+1$ components, f and all derivatives can be calculated simultaneously in one execution. The instruction $u * v$ takes $2n + 1$ scalar multiplications and $n - 1$ scalar additions. This represents a significant reduction

of the computation cost of this multiplication instruction. Similarly, if f evaluates $\texttt{sin}(u)$, the simple- dual implementation computes `sin` and `cos` functions at the same value n times in addition to performing n scalar multiplications. An implementation that uses multivariate duals of dimension $n+1$, calculates the `sin` and `cos` values only once and uses them to multiply with the n derivative components. This leads to significant cost saving as evaluation of trigonometric functions and functions like `sqrt` is expensive. Additionally, an efficient vector-scalar multiplication capability is available for many architectures and can be used to further improve the performance.

A significant reduction of the total operation count achieved by using multivariate duals comes at a cost of increased memory use, which can have a negative impact on performance. For example, if the function f use memory to store m real variables, simple-dual implementation doubles the memory requirement to $2m$. An implementation based on $n+1$ dimensional duals, requires memory for $(n+1)m$ real variables, which can be detrimental for performance or even prohibitive. To optimize performance for a specific application on a specific architecture, one can choose a suitable dimension of multivariate dual numbers. For example, the AD implementation based of multivariate duals of size $n/2+1$ calls the function twice calculating $n/2$ derivatives each time. In this case, some computations repeat but the memory requirements are reduced to $(n/2+1)m$. Such a trade-off can improve AD performance [23].

III. GPU Implementation of Aut Differentiation for Flux Fun

In this section, we review the computation Roe's flux function [24] and its derivatives. Th flux function for FUN3D and is used in ma The approaches for AD implementations of t on a GPU architecture are discussed.

A. Residual Computations on Unstructured G

In three dimensions (3D), FUN3D uses discretization scheme on unstructured mixe that may contain tetrahedra, pyramids, prisms, Independent primitive variables represent d components, and pressure of a fluid and re points. The governing-equation residuals are conservation laws evaluated on a set of med volumes centered at grid points. Edge-based inviscid fluxes are computed at edge midpoints using an approximate Riemann solver. In the current study, Roe's approximate Riemann solver [24] is used. A directed area vector is precomputed for each edge and represents the portion of the control-volume boundary associated with the edge.

B. Flux Function

The 3D Roe's flux function (`fluxkernel`) used in our implementation is based on an open-source Fortran routine available at [25] and is similar to the one used in FUN3D.

Algorithm 1 FLUX(G)

1: Input: Grid G
2: **for** $i \leftarrow 1$ **to** n_{edges} in G **do**
3: $q_L, q_R \leftarrow \texttt{getPrimitive}(G, i)$
4: $dir \leftarrow \texttt{getDirection}(G, i)$
5: $localFlux \leftarrow \texttt{fluxkernel}(q_L, q_R, dir)$
6: $globalFlux(i) \leftarrow localFlux$
7: **end for**
8: **return** $globalFlux$

The correctness of our implementation has been verified by matching inputs and outputs with the FUN3D production routine. The `fluxkernel` routine is called in a loop over edges. The input to the routine is two ("left" and "right") sets of primitive variables, $q_L = (\rho_L, u_L, v_L, w_L, p_L)$ and $q_R = (\rho_R, u_R, v_R, w_R, p_R)$ and a 3D unit vector $dir = (n_x, n_y, n_z)$ representing the direction of the directed area vector associated with the edge. The `fluxkernel` output is a numerical flux vector of size 5 that is post-multiplied with the magnitude of the directed-area vector. A high-level description of the flux computation is shown in Algorithm 1. The input G contains solutions and unstructured-grid metrics.

While the `fluxkernel` routine uses primitive variables as input, the Jacobian employed by the solver for the nonlinear governing equations is the derivative of the discrete residual vector with respect to the vector of conservative variables representing mass, momentum, and energy conservation. There

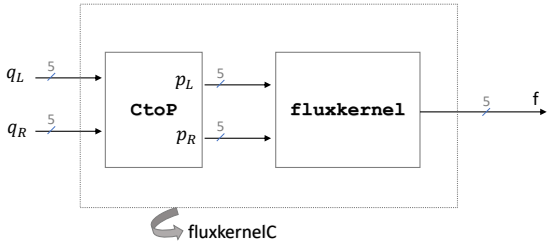

Fig. 1: Flux computation. Q_L, Q_R are conservative variables that are converted to primitive variables q_L, q_R by function `CtoP`. `fluxkernel` uses the primitive variables to compute the flux. GN: I changed primitive notation to q and conservative notation to Q. This needs updated for this figure.

C. Automatic Differentiation of Flux Function

In this paper we study AD approaches for computing Jacobian that is the derivative of the residual vector with respect to conservative variables. There are two ways to account for conversion from conservative to primitive variables. The first way is to hand differentiate the map from conservative to primitive variables, seed the resulting matrix as input, and perform AD of `fluxkernel`. This way is illustrated in

Algorithm 2 FLUXC(G)

1: Input: Grid G
2: **for** $i \leftarrow 1$ **to** n_{edges} in G **do**
3: $Q_L, Q_R \leftarrow$ `getConservative`(G, i)
4: $dir \leftarrow$ `getDirection`(G, i)
5: $localFlux \leftarrow$ `fluxkernelC`(Q_L, Q_R, dir)
6: $globalFlux(i) \leftarrow localFlux$
7: **end for**
8: **return** $globalFlux$

Algorithm 3. The subscript D denotes variables and routines that use duals. The second way that directly performs AD on `fluxkernelC` is illustrated in Algorithm 4. Both Algorithm 3 and Algorithm 4 are written for AD based on simple dual numbers.

Let us first consider Algorithm 3. There are ten `fluxkernel` input parameters representing primitive variables. In the **for** loop in line 3 of Algorithm 3, in iteration j, the derivative of `fluxkernel` is computed with respect to the jth conservative variable. In Line 6, the `seedDual` procedure creates simple dual numbers that are used by the $\texttt{fluxkernel}_\texttt{D}$ function in line 7. For example, for j=1 and k=1, ..., 5, five simple duals are created: $q_{L_D}[k] = \langle q_L[k], \frac{\partial q_L[k]}{\partial Q_L[j]} \rangle$. Here, $q_L[k]$ is obtained by conservative-to-primitive conversion (see the `CtoP` block in Figure 1) and $\frac{\partial q_L[k]}{\partial Q_L[j]}$ is the k, j entry of the hand-differentiated matrix of the conservative-to-primitive map. Note that the $\texttt{fluxkernel}_D$ routine is the `fluxkernel` routine operating on duals and using operator overloading. The $localFlux_D$ is a vector of 5 duals, which is used to update the sparse global Jacobian matrix. As stated earlier, the global Jacobian matrix consists of a block-diagonal matrix Adiag, and an off-diagonal matrix Aoff. Aoff is a block-sparse matrix that is maintained in a block compressed sparse row (CSR) format [26]. The functions in lines 8 and 9 update Aoff and Adiag by accumulating flux derivatives ($localFlux_D$) at Aoff and Adiag locations identified by edge i.

Algorithm 4 is similar to Algorithm 3, except that there is no seeding from a precomputed matrix. In iteration j of **for** loop in line 3, the simple duals are set for the conservative variables (line 6) by initializing the derivative component of the jth variable to 1 and the derivative components of all other variables to zero. These simple duals are then used on line 7 to compute the derivative of `fluxkernelC` with respect to the jth conservative variable.

D. GPU Operator Overload Implementation

We developed several GPU implementations based on simple duals, multivariate duals, and different ways of mapping GPU threads to computations. In the following, we briefly discuss these implementations.

1) Simple duals with ten threads per edge: This implementation is based on simple duals. A DualNum structure has been implemented and supported basic operations through operator overloading. Figure 2 shows the DualNum structure and the multiplication and division operations.

Algorithm 3 FLUX-DERIVATIVE(G, Aoff, Adiag)

1: Input: Grid G
2: **for** $i \leftarrow 1$ **to** n_{edges} in G **do**
3: **for** $j \leftarrow 1$ **to** 10 **do**
4: $Q_L, Q_R \leftarrow$ `getConservative`(G, i)
5: $dir \leftarrow$ `getDirection`(G, i)
6: $q_{L_D}, q_{R_D} \leftarrow$ `seedDual`(Q_L, Q_R, j)
7: $localFlux_D \leftarrow \texttt{fluxkernel}_\texttt{D}(q_{L_D}, q_{R_D}, dir)$
8: `updateAoff`(Aoff, $localFlux_D, G, i$)
9: `updateAdiag`(Adiag, $localFlux_D, G, i$)
10: **end for**
11: **end for**
12: **return** Aoff, Adiag

Algorithm 4 FLUX-DERIVATIVEC(G, Aoff, Adiag)

1: Input: Grid G
2: **for** $i \leftarrow 1$ **to** n_{edges} in G **do**
3: **for** $j \leftarrow 1$ **to** 10 **do**
4: $Q_L, Q_R \leftarrow$ `getConservative`(G, i)
5: $dir \leftarrow$ `getDirection`(G, i)
6: $Q_{L_D}, Q_{R_D} \leftarrow$ `initDual`(Q_L, Q_R, j)
7: $localFlux_D \leftarrow \texttt{fluxkernelC}_\texttt{D}(Q_{L_D}, Q_{R_D}, dir)$
8: `updateAoff`(Aoff, $localFlux_D, G, i$)
9: `updateAdiag`(Adiag, $localFlux_D, G, i$)
10: **end for**
11: **end for**
12: **return** Aoff, Adiag

This implementation exploits parallelism at two levels. The first-level parallelism is across edges, and the second-level parallelism is across iterations for ten partial derivatives ($line$ 2 and $line$ 3 in Algorithm 3 and Algorithm 4). In other words, we assign 10 threads to process an edge. The first-five threads call the edge kernel to compute derivatives with respect to five q_L variables. The next set of five threads computes derivatives with respect to five q_R variables. It is possible to call the same routine for both instances because of the computation symmetry while performing derivatives with respect to q_L and q_R variables. When the `fluxkernel` is called by the second set of five threads, we just need to negate the dir vector as shown in Figure 3.

Listing 1: Illustration of identical `fluxkernel` interface for derivatives with respect to Q_L and Q_R.

```
// set the pointer for the shared memory
double *flux = &fluxes_s[ledge][iter*NDIM];
// for first set of five threads side = 0
// for the next set of five threads side = 1
const double sign = (side == 0) 1.0 : -1.0;
// negative sign to use the same routine due to symmetry
// ql and qr are also swapped
fluxkernel(sign*nx, sign*ny, sign*nz, area, ql, qr, flux);
```

GN: can remove the listing in favor of PDF; not sure how you generated a PDF. I simplified the kernel and notation. BD: change the flag in Figure 3 from "k" to something else. "k" is associated with previously used index through entries of localFlux function. Also nx, ny, and nz are not norms, but components of a unit vector that is direction of the directed area vector. Should change comments in the listing.

The computation has been restructured to minimize the

```
struct DualNum {
  double v_real, v_dual;

  __device__
  DualNum(double real=0,
          double dual=0)
    : v_real(real), v_dual(dual)
  { }

  __device__
  DualNum& operator=(double op1)
  {
   v_real = op1;
   v_dual = 0.0;
   return *this;
  }
};
```

(a) DualNum structure based on simple dual numbers.

```
__forceinline__ __device__ DualNum
operator*(DualNum const& op1,
          DualNum const& op2)
{
  return
  { op1.v_real * op2.v_real,
    op1.v_real * op2.v_dual +
    op1.v_dual * op2.v_real };
}

__forceinline__ __device__ DualNum
operator/(const DualNum &op1,
          const DualNum &op2)
{
  double temp = 1.0 / op2.v_real;
  return { op1.v_real * temp,
    (op1.v_dual* op2.v_real -
     op2.v_dual* op1.v_real)
     * temp * temp};
}
```

(b) DualNum multiplication and division operations.

Fig. 2: DualNum structure and sample operations.

```
// set the pointer for the shared memory
double *flux = &fluxes_s[ledge][iter*NDIM];
// for first set of five threads side = 0
// for the next set of five threads side = 1
const double sign = (side == 0) 1.0 : -1.0;
// negative sign to use the same routine due to symmetry
// ql and qr are also swapped
fluxkernel(sign*nx, sign*ny, sign*nz, area, ql, qr, flux);
```

Fig. 3: Illustration of identical `fluxkernel` interface for derivatives with respect to Q_L and Q_R.

number of temporary variables and reduce the register pressure. The use of shared memory further reduces temporary arrays and variables. Dual local flux vectors $localFlux_D$ are stored in shared memory. Their derivative parts are collectively populated by ten threads. When completed, the matrices of local flux derivatives are copied from the shared memory to the device memory using memory coalescing. Additionally, atomics are used for updating Adiag to avoid race conditions, which can occur when two threads operating on different edges try to write to the same location. Atomics are not required for updating Aoff. However, we found the atomics are faster compared to regular updates on V100. Figure 4 lists the code segment that writes the data from the shared memory to the device memory in a coalesced way.

```
// size of state
constexpr int NDIM = 5;
// size of local Jacobian
constexpr int NDIM2 = NDIM*NDIM;
// number of edges processed by a block
constexpr int NEDGES_BLOCK = 25;
// number of threads per edge
constexpr int NTE = 10;
// NEDGES_BLOCK*NTE threads write NEDGES_BLOCK*NDIM2*2 values
// into the device memory in coalesced way
for (int sid = threadIdx.x; sid < NEDGES_BLOCK*NDIM2*2;
                            sid += NEDGES_BLOCK*NTE)
{
  // set indices for shared and device memory
  .
  .
  .
  // fetch data from shared memory fluxes_s
  double value = fluxes_s[local_edge][side * NDIM2 + id25];
  // first five threads update dervatives values wrt q_L
  // next five threads update dervatives values wrt q_R
  const int id_diag = (side == 0) ? id11 : id22;
  const int id_off  = (side == 0) ? id21 : id12;
  atomicAdd(&a_diag[id_diag * NDIM2 + id25], value);
  atomicAdd(&a_off[id_off * NDIM2 + id25], -value);
}
```

Fig. 4: A block of threads moves values from the shared memory to the device memory in a coalesced way.

Listing 2: A block of threads moves values from the shared memory to the device memory in a coalesced way.

```
// size of state
constexpr int NDIM = 5;
// size of local Jacobian
constexpr int NDIM2 = NDIM*NDIM;
// number of edges processed by a block
constexpr int NEDGES_BLOCK = 25;
// number of threads per edge
constexpr int NTE = 10;
// NEDGES_BLOCK*NTE threads write NEDGES_BLOCK*NDIM2*2
    values
// into the device memory in coalesced way
for (int sid = threadIdx.x; sid < NEDGES_BLOCK*NDIM2*2;
                            sid += NEDGES_BLOCK*NTE)
{
  // set indices for shared and device memory
  .
  .
  .
  // fetch data from shared memory fluxes_s
  double value = fluxes_s[local_edge][side * NDIM2 + id25];
  // first five threads update dervatives values wrt q_L
  // next five threads update dervatives values wrt q_R
  const int id_diag = (side == 0) ? id11 : id22;
  const int id_off  = (side == 0) ? id21 : id12;
  atomicAdd(&a_diag[id_diag * NDIM2 + id25], value);
  atomicAdd(&a_off[id_off * NDIM2 + id25], -value);
}
```

GN: can remove the listing in favor of PDF; not sure how you generated a PDF. I simplified the kernel and notation.

2) Simple duals with five threads per edge: The second implementation is also based on simple duals and uses hierarchical parallelism. However, the second-level parallelism is restricted to five threads per edge. The five threads first compute derivatives with respect to five Q_L variables, and then compute derivatives with respect to five Q_R variables. As in the first implementation, the edge kernel is restructured to minimize the use of temporary variables and uses the shared memory to enable coalesced updates.

3) Simple duals with two threads per edge: This implementation is also based on simple duals and uses hierarchical parallelism. However, the second-level parallelism is restricted

to two threads per edge. One thread is responsible for computing derivatives with respect to five Q_L variables, and the other is responsible for computing derivatives with respect to five Q_R variables. As in the first implementation, the edge kernel is restructured to minimize the use of temporary variables and uses shared memory to enable coalesced updates.

4) Multivariate duals with two threads per edge: This implementation is based on multivariate duals. As explained earlier, multivariate dual numbers enable the simultaneous computation of multiple derivatives. For example, using multivariate dual numbers of size 5, we can compute flux derivatives with respect to five Q_L variables with one call to the AD version of the flux computation routine. We created a DualNum5 structure that uses multivariate duals of size 5 and supports basic operations. Figure 5 shows the DualNum5 structure and a few operations.

E. GPU Operator Overload with Expression Template

Expression templates are a C++ programming technique that builds an expression tree for computation at compile time. They provide the notional convenience of operator overloading with an expectation of efficient code generation. They provide the opportunity for a compiler to generate code only for those parts of an expression that are needed in the computation. For some tasks, such as determining diagonal elements of a product of several large matrices, expression templates result in an optimized code that is much more efficient than any code based even on well-tuned matrix multiplication routines that perform full multiplications. In addition to the option of eliminating unnecessary parts of a calculation, expression templates also provide opportunities for important compiler optimization such as common subexpression elimination and loop fusion. An advanced compiler might be able to take advantage of these opportunities to trade-off between register usage and operation count. The salient features of the expression template implementation, those that support the representation of lazily-evaluated expressions, are shown in figures 6a, 6b, and 7.

Central to the design of expression template implementation are the types that represent expressions. The simplest expression is a dual value; this is represented by the struct $DualNum$, shown in figure 6a. Note that this provides a "function call" operator, $operator()(std :: size\_ti)$, to provide access to the ith element of the dual. Shown in figure 6b is the class template $Prod$ that represents a product of two expressions. We have implemented also Sum, $Diff$ and the other operators needed to support the required calculations. The type $Product\langle DualNum, DualNum\rangle$ represents an expression of the form xy, where both x and y are duals. The type $Product\langle Sum\langle DualNum, DualNum\rangle, DualNum\rangle$ similarly represents $(x + y)z$, where x, y and z are duals. With relatively few building blocks, arbitrary expressions can be represented. The class template $Product$, and its related templates, contain as their data *references* to the expression objects from which is created; they do not evaluate the expressions when constructed. Each class template

```
 1: struct DualNum5 {
 2:   double v, d0, d1, d2, d3, d4;
 3:
 4:   __device__
 5:   DualNum5(double value=0,
 6:             double der0=0,
 7:             double der1=0,
 8:             double der2=0,
 9:             double der3=0,
10:             double der4=0)
11:     : v(value), d1(der1), d2(der2),
12:       d3(der3), d4(der4)
13:   { }
14:
15:   __device__
16:   DualNum5& operator=(double op1)
17:   {
18:     v  = op1;
19:     d0=0.0;
20:     d1=0.0;
21:     d2=0.0;
22:     d3=0.0;
23:     d4=0.0;
24:     return *this;
25:   }
26: };
```

(a) DualNum5 structure.

```
 1: __forceinline__ __device__ DualNum5
 2: operator*(DualNum5 const& op1,
 3:           DualNum5 const& op2)
 4: {
 5:   return
 6:   { op1.v * op2.v,
 7:     op1.v * op2.d0  + op1.d0 * op2.v,
 8:     op1.v * op2.d1 + op1.d1 * op2.v,
 9:     op1.v * op2.d2 + op1.d2 * op2.v,
10:     op1.v * op2.d3 + op1.d3 * op2.v,
11:     op1.v * op2.d4 + op1.d4 * op2.v };
12: }
13:
14: __forceinline__ __device__  DualNum5
15: operator/(DualNum5 const& op1,
16:           DualNum5 const& op2)
17: {
18:   double temp = 1.0 / op2.v;
19:   double temp2 = temp*temp;
20:   return
21:   { op1.v * temp,
22:     (op1.d0 * op2.v - op2.d0 * op1.v) * temp2,
23:     (op1.d1 * op2.v - op2.d1 * op1.v) * temp2,
24:     (op1.d2 * op2.v - op2.d2 * op1.v) * temp2,
25:     (op1.d3 * op2.v - op2.d3 * op1.v) * temp2,
26:     (op1.d4 * op2.v - op2.d4 * op1.v) * temp2 };
27: }
```

(b) DualNum5 multiplication and division operations.

Fig. 5: DualNum5 structure and sample operations.

also provides $operator()()$ which provides access to the ith element of dual resulting from the evaluation of the expression. Only when $operator()()$ is called to access the element does the compiler generate the code required to evaluate that part of the expression. Note that it is in the implementation of $operator()()$ that the mathematical rules for the evaluation of the product of duals is encoded. $DualNum$ also supports creation from, or assignment from, and arbitrary dual number expression; both creation and assignment force the valuation of all the components of the expression. The final piece of the design is illustrated in figure 7. These are the functions, and function template, that overload the multiplication operator to form products of expressions. The critical feature here is the fact that these

```
struct DualNum {
  using value_type = double;
  std::array<value_type, 2> vals;

  __device__
  DualNum(double real=0, double dual=0)
    : vals{real, dual} { }

  template <typename E>
  __device__ DualNum(E const& expr)
    : vals{expr(0), expr(1)} { }

  __device__ value_type
  operator()(std::size_t i) const
  { return vals[i]; }

  __device__ value_type&
  operator()(std::size_t i)
  { return vals[i]; }

  template <typename E>
  __device__ DualNum&
  operator=(E const& expr)
  {
    vals[0] = expr(0);
    vals[1] = expr(1);
    return *this;
  }
};
```

(a) *DualNum* structure with partial expression template support.

```
template <typename LHS, typename RHS>
class Product {
public:
  using value_type = typename LHS::value_type;

  __device__
  Product(LHS const& lhs, RHS const& rhs)
    : lhs(lhs), rhs(rhs) { }

  __device__
  value_type operator()(std::size_t i) const
  { return i == 0
      ? lhs(0) * rhs(0)
      : lhs(0) * rhs(1) + lhs(1) * rhs(0);
  }

private:
  LHS const& lhs;
  RHS const& rhs;
};
```

(b) Example DualNum lazily-evaluated multiplication support with expression templates.

Fig. 6: Expression template types representing a dual number and a lazily-evaluated product.

operators create, but do not evaluate, the objects representing the expressions.

IV. Results and Performance Evaluation

We evaluated the performance of various implementations listed in Table I on the NVIDIA Tesla V100-PCIE-16GB, Memory Clock Rate (KHz): 877000, and Peak Memory Bandwidth (GB/s): 898. The compilation of code was done using CUDA 11.4 with gcc/10.2. The test case used here is based on transonic turbulent flow over the semispan wing-body [27] configuration shown in Fig. 8. The freestream Mach number is 0.85, the angle of attack is zero degrees, and the Reynolds number, based on the mean aerodynamic chord, is 5 million. The computational mesh consists of 1.1 million grid vertices, 1.2 million prisms, 3.0 million tetrahedra, and 7.3 thousand

```
__device__ inline DualNum
operator*(DualNum const& x, double y)
{
  return x * DualNum(y);
}

__device__ inline DualNum
operator*(double x, DualNum const& y)
{
  return DualNum(x) * y;
}

template <typename EXPR>
__device__ auto
operator*(EXPR const& x, double y)
{
  DualNum temp{x};
  return temp * y;
}

template <typename EXPR>
__device__ auto
operator*(double x, EXPR const& y)
{
  DualNum temp{y};
  return x * temp;
}
```

Fig. 7: Free function support for multiplication of expressions involving dual numbers.

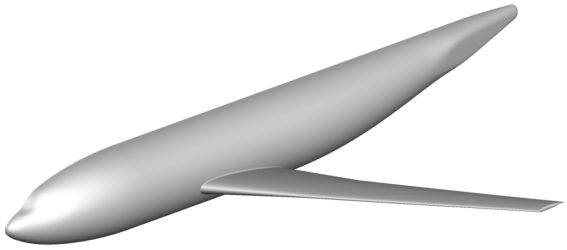

Fig. 8: Wing-body configuration taken from Ref. [27].

pyramids. The total number of edges in the mesh is 5.97 million. This problem size is representative of the workload that would typically be placed on a single GPU in practice. For the purposes of the current study, the input/output to the automatic derivative routine is extracted from an arbitrary time step during the nonlinear convergence of the mean flow equations.

The execution time of the various implementations is listed in Table II. For comparison, we include the timing of a hand-differentiated code and Tapenade. Note that somewhat unexpectedly, dn5_2 implementation outperforms the hand-differentiated code[3]

We looked at the operation count for various overload operator-based implementations using dual numbers and dual numbers with expression templates. Note that all the computation for the automatic differentiation is done in double precision. For updates to *Aoff*, which is a single precision array, the double precision values are converted to single precision just before getting written to the device memory.

[3]The hand-differentiated code attempts to compute all partial derivatives in one routine and thereby increases the register pressure. The hand-differentiated code can be improved using the lessons learned from the current study.

TABLE I: Designation for various implementations.

Designation	Approach
dn_10	DualNum struct with ten threads per edge
dn_5	DualNum struct with five threads per edge
dn_2	DualNum struct with two threads per edge
dn5_2	DualNum5 struct with two threads per edge
hd	Hand-differentiated code[1]
tp	Tapenade source code transformation[2]
edn_5	Expression template with dual numbers based on dn_5
edn5_2	Expression template with dual numbers based on dn5_2

[1]Hand-differentiated code is optimized FUN3D production code.

[2]Source code transformation applied to FUN3D production code.

TABLE II: Execution time in milliseconds and relative slowdown for various implementations on V100. The relative slowdown is computed with reference to the performance of the hand-differentiated code.

Approach	Execution Time (ms)	Relative Slowdown
hd	8.1	1.0
tp	22.8	2.8
dn_10	13.16	1.62
dn_5	8.99	1.11
dn_2	12.18	1.50
dn5_2	7.80	0.96
edn_5	10.32	1.27
edn5_2	8.87	1.09

As expected, we observed differences in the operation count for implementations based on simple and multivariate duals. There was also variation in the operation count depending on the version of algorithm used: Algorithm 3 or Algorithm 4. Note that in Algorithm 4, we modify the edge kernel to accept conservative variables as input, and for Algorithm 3 we need to seed the input with a hand-differentiated matrix. However, surprisingly the implementation using expression templates has a higher count of FLOPs than the implementations that use simple and multivariate dual numbers. The results are summarized in Table III.

TABLE III: Number of double precision fused multiply and add (DFMA), double precision add (DADD), and double precision multiplication (DMUL) for various implementations on V100.

Approach	Seeding	DFMA (10^9)	DADD (10^9)	DMUL (10^9)	Total FLOPS (10^9)
dn_10	Yes	9.25	2.82	7.75	29.07
dn_5	Yes	8.95	2.82	7.72	28.45
dn_2	Yes	8.95	2.82	7.70	28.43
dn5_2	No	4.79	2.15	4.54	16.26
edn_5	Yes	11.16	5.32	9.27	36.92
edn5_2	No	5.91	3.71	4.11	19.64

A. *Performance Analysis for dn5_2*

We analyzed the best performing implementation, dn5_2, to see if it can be improved further. The atomic updates to *Aoff* and *Adiag* create the major traffic to the device memory. To get an estimate on how much we are spending on this part, first, we disabled the atomic updates[4]. Next, we disabled the automatic differentiation routine and measured only the time spent on the atomic updates. These timings along with the regular execution time are summarized in Table IV. The timings indicate that in dn5_2, we spend most of the time in atomic updates, and the computation part is almost *free*. Hence, if we want to make any improvement first, we need to see if we can improve the performance of atomic updates and then try to improve the computation part. We profiled the code with NVIDIA NSight Compute to explore how close we are to the possible peek of compute and memory throughput.

TABLE IV

Time(ms)	Code segment active
7.80	Full code
6.85	Computation with no atomic updates[5]
7.57	Only atomic updates

The roofline model and workload analysis from the profiler are shown in Figure 9. The arithmetic intensity of the dn5_2 kernel is 3.69 and the performance of the kernel is 2.016 TFLOPS/s. The kernel is memory bound, as indicated in the roofline chart, and the overall performance is close to the peak. From the profiler we observed the device memory bandwidth of 610 GB/s, which is around 61% of the peak device memory bandwidth. The workload analysis indicates that the FP64 utilization is close to 75% of the peak FP64 throughput. If we examine the application bandwidth requirement, which is the minimum data that needs to be moved between memory and the SMs, we are only achieving 435 GB/s from the application. However, we are forced to access more data than required as our coalesced reads and writes are of size $25 * 4$ bytes or $25 * 8$ bytes. The cache line size is of 128 bytes resulting in additional memory accesses. Additionally, the 5×5 blocks that we are accessing in the kernel are not contiguous and that also slows down the memory sub-unit. Because of this we believe, it is difficult to improve the performance further, and we are almost performing as well as we could.

V. Conclusion

We presented several GPU implementations of forward mode AD for a commonly used flux computation function in CFD. We identified common challenges faced in obtaining a highly-performant GPU implementation and presented solutions to address them and experimented with different techniques to improve the performance. Our experimentation led to

[4]We need to be careful on how to disable the atomic updates because the compiler can optimize *away* a lot of code. We typically do this by putting the code segment to be disabled inside an if condition that is evaluated at runtime.

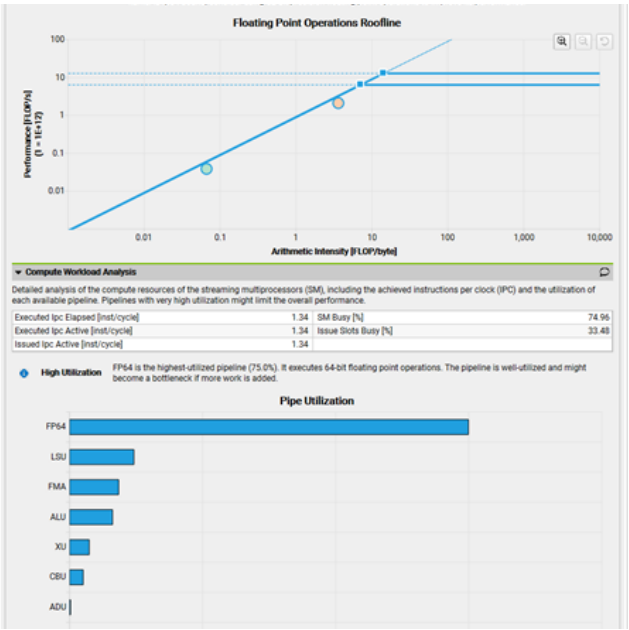

Fig. 9: Roofline model and workload analysis. [27].

a GPU implementation that outperforms all other existing GPU implementations. The analysis of this GPU implementation leads us to believe that it will be hard to improve for this kernel. However, we believe that the techniques used here can be utilized more broadly in design of fast GPU implementations for other scientific (especially CFD) computations.

ACKNOWLEDGMENT

We should acknowledge Eleni Adam's help with expression templates implementation.

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks ...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

[1] E. J. Nielsen and B. Diskin, "Discrete Adjoint-Based Design Optimization of Unsteady Turbulent Flows on Dynamic Overset Unstructured Grids," *AIAA Journal*, vol. 51, no. 6, pp. 1355–1373, 2013. [Online]. Available: https://doi.org/10.2514/1.J051859

[2] A. Griewank and A. Walther, *Principles and Techniques of Algorithmic Differentiation*, 2nd ed. Society for Industrial and Applied Mathematics, 2008.

[3] J. C. Newman, W. K. Anderson, and D. L. Whitfield, "Multidisciplinary Sensitivity Derivatives Using Complex Variables," Computational Fluid Dynamics Laboratory, NSF Engineering Research Center for Computational Field Simulation, Mississippi State Univ., Tech. Rep. MSSU-COE-ERC-98-08, July 1998. [Online]. Available: https://fun3d.larc.nasa.gov/papers/MsReport.pdf

[4] J. N. Lyness and C. B. Moler, "Numerical Differentiation of Analytic Functions," *SIAM Journal on Numerical Analysis*, vol. 4, no. 2, pp. 202–210, 1967. [Online]. Available: https://doi.org/10.1137/0704019

[5] J. Lyness, "Numerical algorithms based on the theory of complex variables," in *Proceedings of 22nd ACM National Conference*. Washington, DC, USA: Thomas Book Co., 1967, p. 124–134. [Online]. Available: https://doi.org/10.1145/800196.805983

[6] *CppAD:A C++ Algorithmic Differentiation Package*, last accessed 6/23/23. [Online]. Available: https://cppad.readthedocs.io/en/latest/user_guide.html

[7] U. Naumann, J. Lotz, K. Leppkes, and M. Towara, "Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations," *ACM Trans. Math. Softw.*, vol. 41, no. 4, oct 2015. [Online]. Available: https://doi.org/10.1145/2700820

[8] A. Walther, "Getting started with ADOL-C," in *Combinatorial Scientific Computing, 01.02. - 06.02.2009*, ser. Dagstuhl Seminar Proceedings, U. Naumann, O. Schenk, H. D. Simon, and S. Toledo, Eds., vol. 09061. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2009/2084/

[9] C. H. Bischof, P. D. Hovland, and B. Norris, "Implementation of Automatic Differentiation Tools," *SIGPLAN Not.*, vol. 37, no. 3, p. 98–107, jan 2002. [Online]. Available: https://doi.org/10.1145/509799.503047

[10] A. Griewank and A. Walther, "Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation," *ACM Trans. Math. Softw.*, vol. 26, no. 1, p. 19–45, mar 2000. [Online]. Available: https://doi.org/10.1145/347837.347846

[11] R. J. Hogan, "Fast Reverse-Mode Automatic Differentiation Using Expression Templates in C++," *ACM Trans. Math. Softw.*, vol. 40, no. 4, jul 2014. [Online]. Available: https://doi.org/10.1145/2560359

[12] H. M. Bücker and G. F. Corliss, "A Bibliography on Automatic Differentiation," in *Automatic Differentiation: Applications, Theory, and Implementations*, ser. Lecture Notes in Computational Science and Engineering, H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, Eds. New York, NY: Springer, 2005, vol. 50, pp. 321–322.

[13] J. Hückelheim, N. Kukreja, S. H. K. Narayanan, F. Luporini, G. Gorman, and P. Hovland, "Automatic Differentiation for Adjoint Stencil Loops," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337906

[14] L. Hascoet and V. Pascual, "The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification," *ACM Trans. Math. Softw.*, vol. 39, no. 3, may 2013. [Online]. Available: https://doi.org/10.1145/2450153.2450158

[15] J. Hückelheim, M. Schanen, S. H. K. Narayanan, and P. Hovland, "Vector forward mode automatic differentiation on simd/simt architectures," in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3404397.3404470

[16] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme," in *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476165

[17] E. Phipps, R. Pawlowski, and C. Trott, "Automatic Differentiation of C++ Codes on Emerging Manycore Architectures with Sacado," *ACM Trans. Math. Softw.*, vol. 48, no. 4, dec 2022. [Online]. Available: https://doi.org/10.1145/3560262

[18] M. E. Schüle, M. Springer, A. Kemper, and T. Neumann, "LLVM Code Optimisation for Automatic Differentiation: When Forward and Reverse Mode Lead in the Same Direction," in *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*, ser. DEEM '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3533028.3533302

[19] I. Ifrim, V. Vassilev, and D. J. Lange, "GPU Accelerated Automatic Differentiation With Clad," *Journal of Physics: Conference Series*, vol. 2438, no. 1, p. 012043, feb 2023. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/2438/1/012043

[20] J. Blühdorn, N. R. Gauger, and M. Kabel, "AutoMat: automatic differentiation for generalized standard materials on GPUs," *Computational Mechanics*, vol. 69, no. 2, pp. 589–613, nov 2021. [Online]. Available: https://doi.org/10.1007/s00466-021-02105-2

[21] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell,

S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3097283

[22] W. K. Anderson, R. T. Biedron, J.-R. Carlson, J. M. Derlaga, C. T. D. Jr., P. A. Gnoffo, D. P. Hammond, K. E. Jacobson, W. T. Jones, B. Kleb, E. M. Lee-Rausch, G. C. Nastac, E. J. Nielsen, M. A. Park, C. L. Rumsey, J. L. Thomas, K. B. Thompson, A. C. Walden, L. Wang, S. L. Wood, W. A. Wood, B. Diskin, Y. Liu, and X. Zhang, *FUN3D Manual 14.0.1*, NASA/TM-2023-0004211, 2023.

[23] J. Revels, M. Lubin, and T. Papamarkou, "Forward-Mode Automatic Differentiation in Julia," 2016.

[24] P. P. Roe, "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, vol. 43, no. 2, p. 357–372, 1981. [Online]. Available: https://doi.org/10.1016/0021-9991(81)90128-5

[25] K. Masatsuka, "I Do Like CFD, Vol. 1. Governing Equations and Exact Solutions. Second Edition." 2013. [Online]. Available: http://www.cfdbooks.com/

[26] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[27] K. R. Laflin, S. M. Klausmeyer, T. Zickuhr, J. C. Vassberg, R. A. Wahls, J. H. Morrison, O. P. Brodersen, M. E. Rakowitz, E. N. Tinoco, and J.-L. Godard, "Data Summary from Second AIAA Computational Fluid Dynamics Drag Prediction Workshop," *AIAA Journal of Aircraft*, vol. 42, no. 5, pp. 1165 – 1178, 2005.