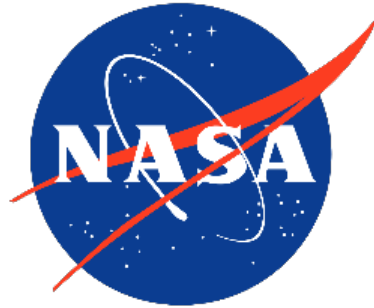


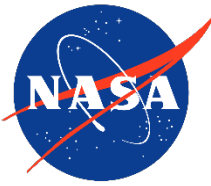
Performance Optimization of Automatic Differentiation for Unstructured-Grid CFD Applications on Emerging Heterogeneous Architectures



Gabriel Nastac, Aaron Walden, Eric Nielsen, and Boris Diskin
NASA Langley Research Center

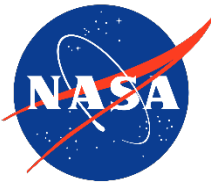
Mohammad Zubair, and Desh Ranjan
Old Dominion University

Motivation



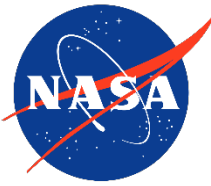
- Many scientific and engineering applications require repeated calculations of derivatives of output functions with respect to input parameters
- In CFD, derivatives of flux functions with respect to state variables are needed for computing Jacobians
- Traditionally, derivatives are computed by hand differentiation
 - Manual, labor-intensive process that is prone to error
- Automatic Differentiation (AD) is an alternative methodology that automates accurate derivative calculation and can significantly speed up code development

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. Paper presented at the IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC), 11 pp. doi: 10.1109/HiPC58850.2023.00055



Automatic Differentiation (AD)

- We are interested in AD for functions that have been coded as a computer program, which executes a sequence of simple arithmetic operations and functions including if and loop structures
- The chain rule can be repeatedly applied to a sequence of elementary operations to compute the partial derivative of the final output with respect to the input variables
- Forward Mode: The complexity of an efficient forward-mode AD implementation is proportional to the number of input parameters and is suitable for applications where there are few input parameters
- Reverse-mode AD has complexity proportional to the number of output functions and is suitable for applications where there are many input parameters and few output functions



Forward Mode AD using Dual Numbers

A simple dual number (or dual) is an ordered pair of real numbers. Arithmetic operations are extended to dual numbers. For example, the result of a multiplication of a dual number $(4, 3)$ with another dual number $(2, 5)$ is $(8, 26)$ and can be represented as:

$$(4 + 3\epsilon) * (2 + 5\epsilon) = 4 * 2 + (3 * 2 + 4 * 5)\epsilon = 8 + 26\epsilon, \quad \text{with } \epsilon^2 = 0$$

$f(x) = x^2 + x$, compute derivative with respect to x at $x = 3.0$

Represent x as a dual number with derivative component 1.

$$f'(x) = (3 + 1\epsilon) * (3 + 1\epsilon) + (3 + 1\epsilon) = 12 + 7\epsilon, \quad \text{with } \epsilon^2 = 0$$

NOTE: The first component is the value of the function and the second component has the derivative of the function.

$f(x, y) = x^2 + y^2$, compute derivative at $x = 2$ and $y = 3$

Compute partial derivative with respect to x , with x as $(2 + 1\epsilon)$ and y as $(3 + 0\epsilon)$

$$(2 + 1\epsilon) * (2 + 1\epsilon) + 3 * 3 = 13 + 4\epsilon$$

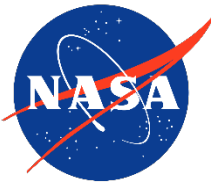
$$\frac{\partial f(x, y)}{\partial x} = 2x$$

Compute partial derivative with respect to y , with x as $(2 + 0\epsilon)$ and y as $(3 + 1\epsilon)$

$$2 * 2 + (3 + 1\epsilon) * (3 + 1\epsilon) = 13 + 6\epsilon$$

$$\frac{\partial f(x, y)}{\partial y} = 2y$$

Simple Dual Number Structure

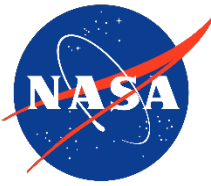


```
1: struct DualNum {
2:     double v_real, v_dual;
3:
4:     __device__
5:     DualNum(double real=0,
6:             double dual=0)
7:         : v_real(real), v_dual(dual)
8:     { }
9:
10:    __device__
11:    DualNum& operator=(double op1)
12:    {
13:        v_real = op1;
14:        v_dual = 0.0;
15:        return *this;
16:    }
17: };
```

(a) DualNum structure based on simple dual numbers.

```
1: __forceinline__ __device__ DualNum
2: operator*(DualNum const& op1,
3:           DualNum const& op2)
4: {
5:     return
6:     { op1.v_real * op2.v_real,
7:       op1.v_real * op2.v_dual +
8:       op1.v_dual * op2.v_real };
9: }
10:
11: __forceinline__ __device__ DualNum
12: operator/(const DualNum &op1,
13:          const DualNum &op2)
14: {
15:     double temp = 1.0 / op2.v_real;
16:     return { op1.v_real * temp,
17:             (op1.v_dual* op2.v_real -
18:              op2.v_dual* op1.v_real)
19:             * temp * temp};
20: }
```

(b) DualNum multiplication and division operations.



AD using Multivariate Dual Numbers

$f(x, y) = x^2 + y^2$, compute derivative at $x = 2$ and $y = 3$

A multivariate dual number for two variables, is a vector of three components. The first component is the value and the second and third components are the derivative with respect to the variables

Compute partial derivative with respect to x , and y with x as $(2 + 1\epsilon_1 + 0\epsilon_2)$ and y as $(3 + 0\epsilon_1 + 1\epsilon_2)$

$$(2 + 1\epsilon_1 + 0\epsilon_2) * (2 + 1\epsilon_1 + 0\epsilon_2) + (3 + 0\epsilon_1 + 1\epsilon_2) * (3 + 0\epsilon_1 + 1\epsilon_2) = 13 + 4\epsilon_1 + 6\epsilon_2 \quad \text{with } \epsilon_1^2, \epsilon_2^2, \epsilon_1\epsilon_2 = 0$$


$$\frac{\partial f(x, y)}{\partial x} = 2x$$

$$\frac{\partial f(x, y)}{\partial y} = 2y$$

NOTE: When we use a multivariate dual number for two variables, we make a single pass to evaluate the function with multivariate dual numbers as opposed to two function evaluations in the case of simple dual numbers. The tradeoff is the additional memory requirement to support multivariate dual numbers and their operations

Multivariate Dual Number Structure

```

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), d0(der0), d1(der1),
12:       d2(der2), 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: };
27:

```

(a) DualNum5 structure.

```

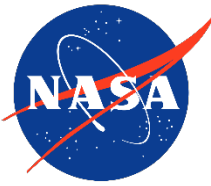
1: __forceinline__ __device__ DualNum5
2: operator*(DualNum5 const& op1,
3:           DualNum5 const& op2)
4: {
5:     return
6:     { op1.v * op2.v,
7:       op1.v * op2.d0 + op1.d0 * op2.v,
8:       op1.v * op2.d1 + op1.d1 * op2.v,
9:       op1.v * op2.d2 + op1.d2 * op2.v,
10:      op1.v * op2.d3 + 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: }

```

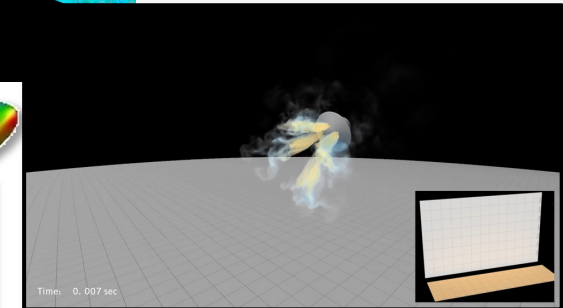
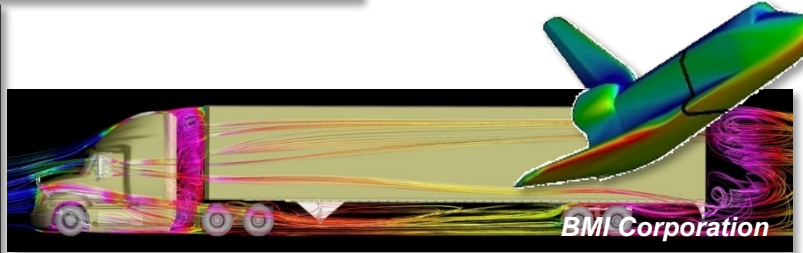
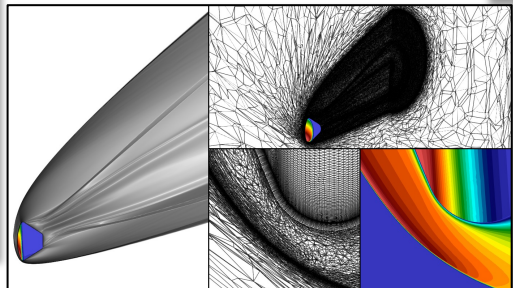
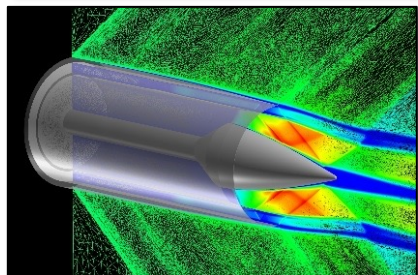
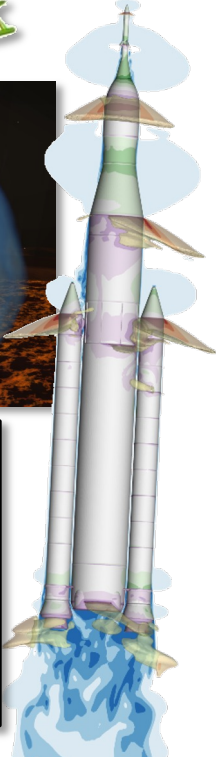
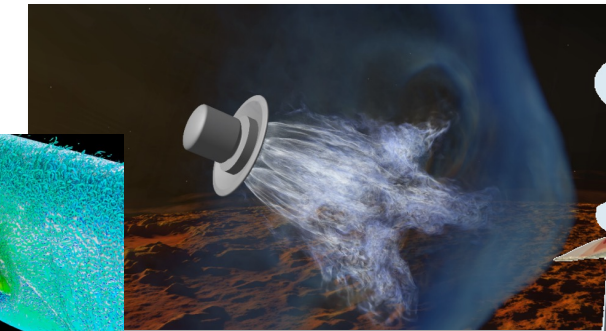
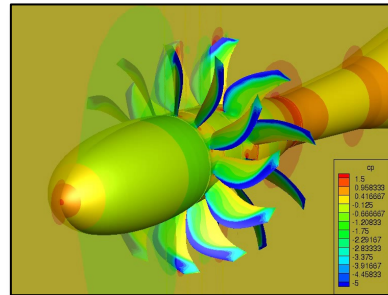
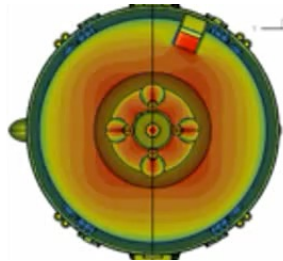
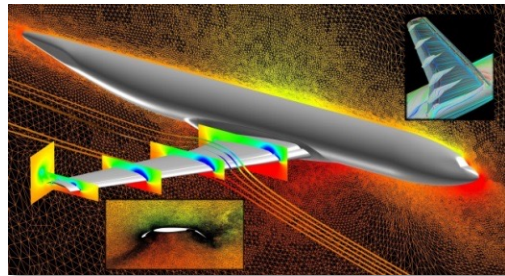
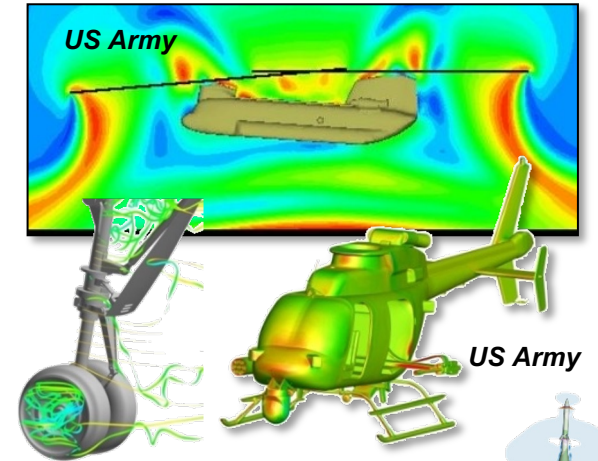
(b) DualNum5 multiplication and division operations.

Fully Unstructured Navier-Stokes 3D (FUN3D)

<https://fun3d.larc.nasa.gov>



- Solves the time-dependent compressible Navier-Stokes equations in fully implicit form on unstructured grids with thermochemical nonequilibrium and assorted turbulence treatments
- Combinations of rigid / overset / deforming grids and 6DOF
- Aeroelastic modeling using mode shapes, full FEM, etc.
- **Constrained / multipoint adjoint-based design**
- Automated grid adaptation – from CAD
- Capabilities fully integrated, online documentation, training videos, tutorials
- **Lightweight abstraction over CUDA C++ enables performance across NVIDIA / AMD / Intel GPUs and multicore CPUs**

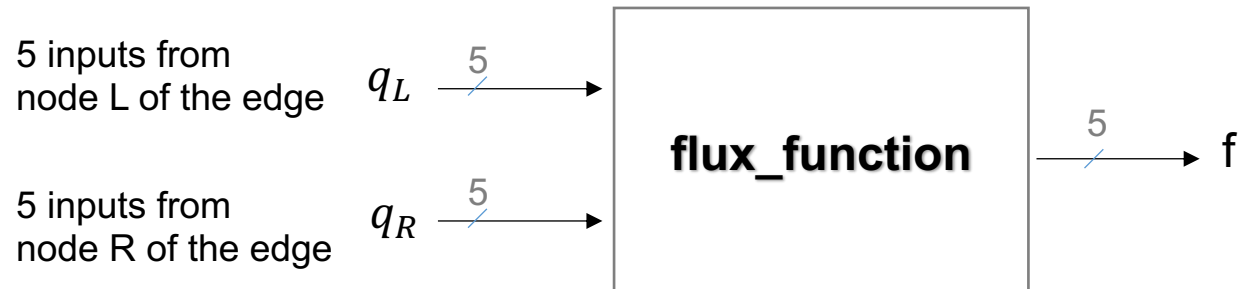
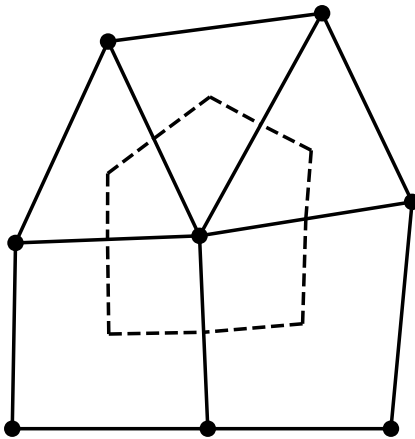


CFD Application

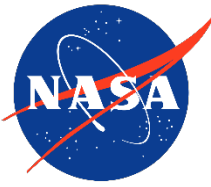


Simplified Version. Execute *flux_function* for each edge (dual-face) in the unstructured grid. A single execution of the *flux_function* computes the flux associated with an edge. The *flux_function* takes ten real inputs and produces five real outputs

The calculation of the flux at an edge is independent of flux calculations at other edges. Thus, the residual computation is parallelized over edges. However, care must be taken to avoid race conditions, which occur when two threads operating on different edges write to the same location corresponding to a residual at a grid point



Grid nomenclature: Solid lines denote primal grid elements while dual volumes are denoted by dashed lines.

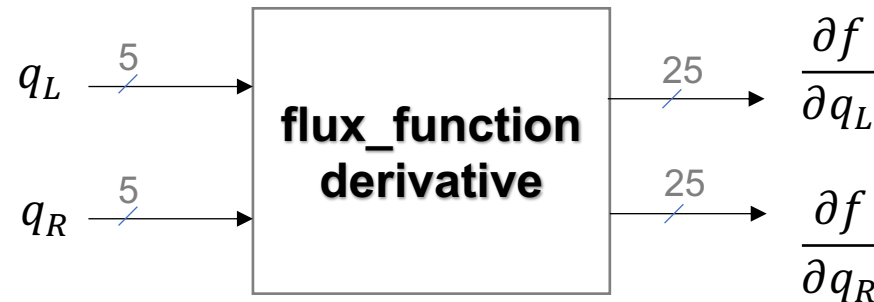


Jacobian Assembly

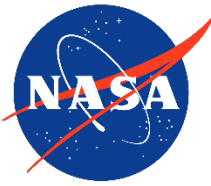
The global Jacobian matrix represents the derivative of the residual vector with respect to the vector of solution components. The Jacobian assembly is also parallelized over edges. At each edge, the derivatives of the flux vector are computed with respect to the inputs of the *flux_function*. The resulting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

The derivative of the *flux_function* locally generates an output of size $2 \times 5 \times 5$ that is used to build the global Jacobian matrix. As before, care must be taken to avoid race conditions

Note: The *flux_function* is templated and reused for many purposes (residual, Jacobian, BCs, adjoint). Also, the syntax is equivalent to standard arrays (e.g., `dual ql[5]` versus `double ql[5]`), preventing the need for substantially changing code in the existing code base

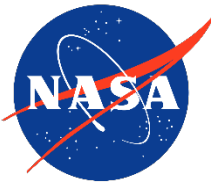


Performance Challenges



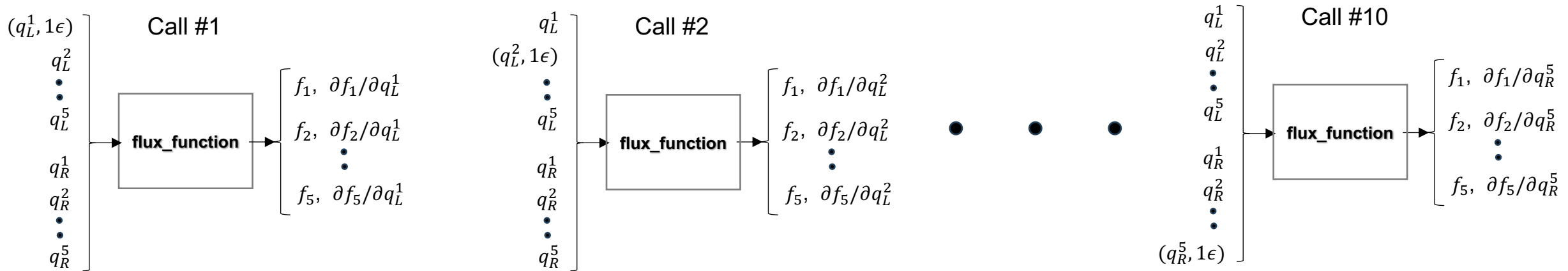
There are several challenges for developing a high-performance Jacobian assembly implementation:

- The flux computation within the *flux_function* involves many intermediate variables that create high register pressure and require 5 memory updates per edge. The derivative (Jacobian) computations further increase 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 *flux_function* locally generates an output of size $2 \times 5 \times 5$ 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 while Aoff is stored with single precision
 - The two 5×5 local Jacobian blocks are added to two 5×5 blocks of the Adiag matrix and two 5×5 blocks of the Aoff matrix



AD of *flux_function* using Simple Dual Numbers

Call *flux_function* ten times with input initialized as dual numbers. For the i th call, set the derivative component of the i th input as one.



All *flux_function* calls are independent and can be executed in parallel, thus exploiting second-level parallelism. Note that the first-level parallelism is across edges in the grid. We explored several mappings of GPU threads to process an edge

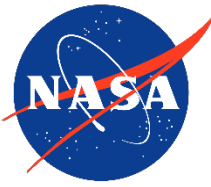
dn_10: 10 threads/edge, spread all 10 calls across 10 threads. Assign a warp of 32 threads to process three edges (two threads inactive)

dn_5: 5 threads/edge, spread 5 calls across 5 threads (2 calls/thread). Assign a warp of 32 threads to process 6 edges

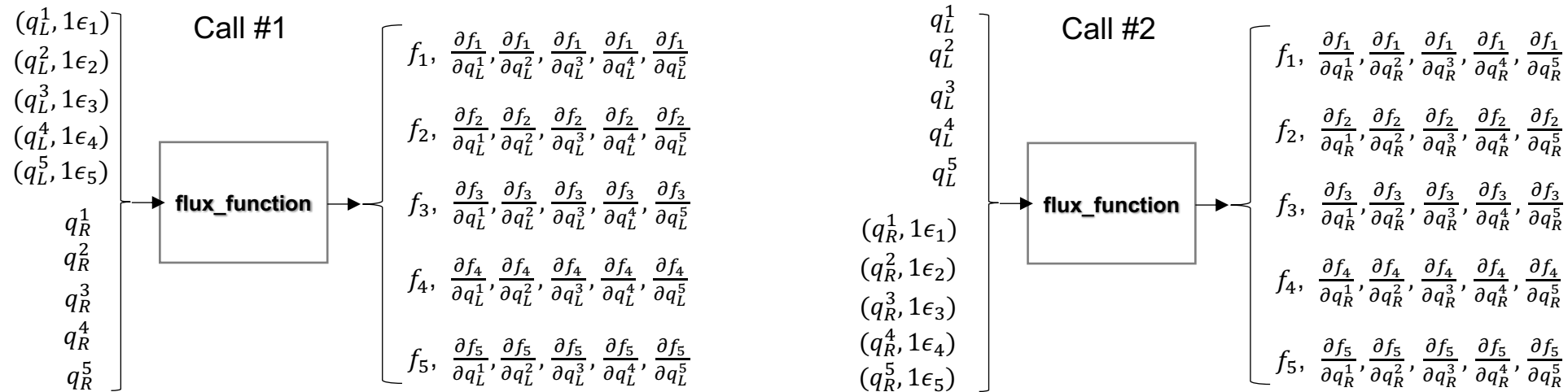
dn_2: 2 threads/edge with each thread making five calls and computing derivatives w.r.t q_L and the other w.r.t q_R . (16 edges/warp)

Other Optimizations: Use of shared memory to make updates to global Jacobian efficient and reduce register pressure

AD of flux_function using Multivariate Dual Number of Dimension 5

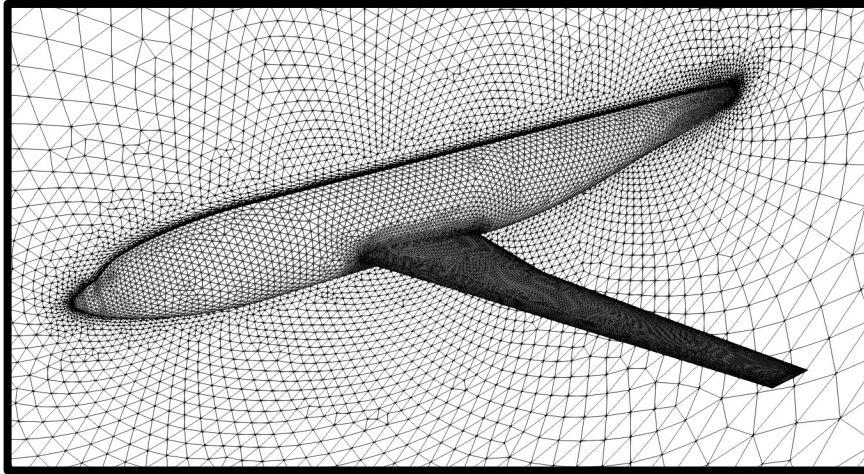
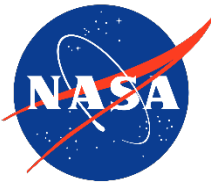


Call *flux_function* two times with input initialized as multivariate dual numbers of dimension 5.



The two *flux_function* calls are independent and can be executed in parallel,
dn5_2: use 2 threads/edge with each thread making one call and computing derivatives w.r.t q_L and the other w.r.t q_R . (16 edges/warp)
 Other Optimizations: Use of shared memory to make updates to global Jacobian efficient and reduce register pressure

Performance Comparison



DPW2 airplane case

(<https://dpw.larc.nasa.gov/DPW2/>)

- 1.1M points
- 4.2M mixed-elements
- 6.0M edges/dual-faces

TABLE I: Designation for the tested implementations.

Designation	Approach
hd	Hand-differentiated code ¹
tp	Tapenade source code transformation ²
enzyme	Enzyme with two threads per edge
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
edn_5	Expression template with dual numbers based on dn_5
edn5_2	Expression template with dual numbers based on dn5_2

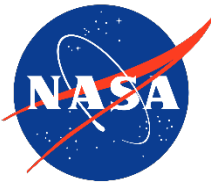
¹Hand-differentiated code is optimized FUN3D production code.

²Source 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	8.10	1.00
tp	22.80	2.80
enzyme	9.10	1.12
dn_10	13.16	1.62
dn_5	12.18	1.50
dn_2	8.99	1.11
dn5_2	7.80	0.96
edn_5	10.32	1.27
edn5_2	8.87	1.09

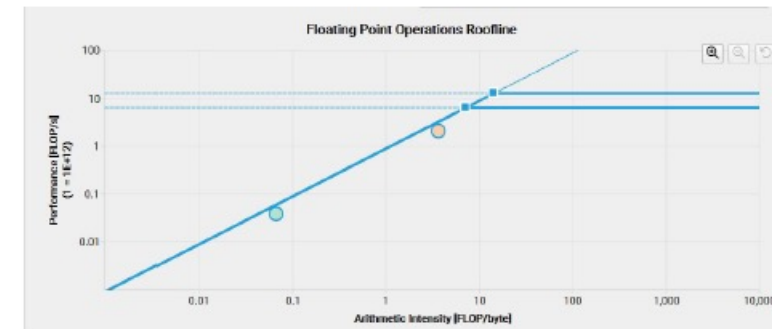
Performance Analysis for dn5_2



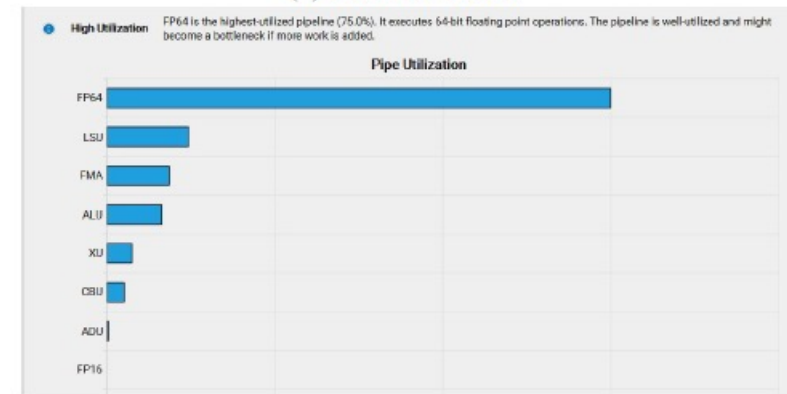
The memory updates to Aoff and Adiaq cause the majority of the traffic to global device memory.

TABLE IV: Computation Vs. memory update times.

Time(ms)	Code segment active
7.80	Full code
7.57	Only memory updates with no computation



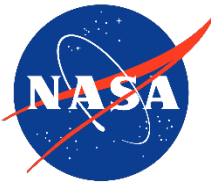
(a) Roof line model.



(b) Workload analysis.

Fig. 9: The roofline model and workload analysis from the profiler

Summary



- We considered several implementations of forward-mode Automatic Differentiation (AD) for a common flux function used in a practical computational fluid dynamics tool
- We experimented with different existing and newly developed techniques to improve AD performance on GPUs. Our experimentation led to an AD implementation based on multivariate dual numbers of optimal dimension. This implementation outperforms all other GPU implementations considered
- The analysis of this AD implementation indicated that its performance on NVIDIA V100 GPU is close to peak performance and would be difficult to improve further
- The AD methodology developed in this work is for a 5x5 block size that corresponds to the Navier-Stokes equations and is fundamental to most CFD applications. Hence, a large number of CFD applications would potentially benefit from the AD methodology developed in this paper
 - Approach is general and applicable to arbitrarily sized blocks
 - Extension to GPU-capable AD adjoint capability is in progress

This effort is sponsored by the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We thank Eleni Adam (Old Dominion University) for helping with the GPU implementation of expression templates for simple and multivariate duals.