A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs

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

Boris Diskin
National Institute of Aerospace
Hampton, Virginia

Mohammad Zubair
Old Dominion University
Norfolk, Virginia

***Abstract*—This paper presents a new mixed-precision implementation of a linear-solver kernel used in practical large-scale CFD simulations to improve GPU performance. The new implementation reduces memory traffic by using the half-precision format for some critical computations while maintaining double-precision solution accuracy. As the linear-solver kernel is memory bound on GPUs for practical CFD applications, a reduction in memory traffic directly translates to improved performance. The performance of the new implementation is assessed for a benchmark steady flow simulation and a large-scale unsteady turbulent flow application. Both studies were conducted using NVIDIA® Tesla V100 GPUs on the Summit system at the Oak Ridge Leadership Computing Facility.**

***Index Terms*—mixed precision, high performance computing, optimization, GPU**

I. Introduction

The relations between velocity, pressure, density, and temperature of a moving fluid are described by the Navier-Stokes (NS) equations. The NS equations constitute a system of time-dependent nonlinear partial differential equations (PDEs) expressing the conservation of mass, momentum, and energy and are characterized by tightly-coupled multiscale interactions. The system is often closed using auxiliary PDEs governing turbulence quantities. Accurate and efficient simulations of aerodynamic flows are challenging and require significant computational resources. FUN3D is a suite of computational fluid dynamics (CFD) software [1] developed at the NASA Langley Research Center to solve the NS equations on unstructured grids for a broad range of applications across the speed range [2], [3].

A system of nonlinear flow equations can be formally represented as

$$\mathbf{R}\left(\mathbf{q}\right) = 0 \tag{1}$$

where $\mathbf{q}$ is the solution vector. The discrete operator $\mathbf{R}\left(\mathbf{q}\right)$ may, for example, represent a discretization of the steady-state Reynolds-Averaged Navier Stokes (RANS) equations, where the meanflow NS equations may or may not be tightly coupled with the turbulence closure equations. The nonlinear iterations are based on a correction scheme

$$\frac{V}{\Delta\tau}\mathbf{\Delta q} + \frac{\partial\hat{\mathbf{R}}}{\partial\mathbf{q}}\mathbf{\Delta q} = -\mathbf{R}\left(\mathbf{q^n}\right) \tag{2}$$

$$\mathbf{q^{n+1}} = \mathbf{q^n} + \mathbf{\Delta q} \tag{3}$$

Here, $\mathbf{q^{n+1}}$ and $\mathbf{q^n}$ are the solutions at iterations $n+1$ and n, respectively; $\frac{\partial\mathbf{R}}{\partial\mathbf{q}}$ is an approximation to the Jacobian; V is a median-dual control volume; and $\Delta\tau$ is a pseudo-time step. An approximate nearest-neighbor Jacobian for the meanflow equations is formed at each control volume using a linearization of first-order inviscid fluxes and second-order viscous fluxes.

The approximate Jacobian is a large system of block-sparse linear equations. The block size of the linearization matrix is determined by the number of governing equations and may range from one to several dozen in the case of general gas mixtures. For perfect-gas simulations, the system of equations describing the meanflow evolution in three dimensions is composed of five equations. The linearization matrix derived from these equations consists of 5×5 blocks. Here, multicolor point-implicit iterations are used to solve the system of linear equations. Compared to widely used Jacobi iterations [4], multicolor iterations offer better iterative convergence, similar opportunities to overlap communication and computation, and comparable concurrency: the solution at grid vertices of a color can be updated simultaneously.

The overall execution time of the FUN3D linear solver is dominated by a block-sparse matrix-vector multiply operation. Due to the operation's low arithmetic intensity (≈ 0.5), performance is bound by main memory bandwidth on modern Central Processing Units (CPUs) and Graphical Processing Units (GPUs). The primary challenges in achieving good performance for the linear solver result from the nature of unstructured grids: irregular memory accesses and a variable number of non-zero blocks in each row of the matrix. The need for high memory bandwidth and high concurrency make the linear solver a suitable application for GPUs. In earlier work, we reported an optimized implementation of the linear-solver kernel on GPUs [5]. An optimized block-sparse matrix-vector kernel achieves close to 90% of memory bandwidth on the NVIDIA® Tesla V100 GPU. This results in a GPU speedup of 3.5-5.0× over the current generation of dual-socket CPUs. Even with optimized block-sparse matrix-vector operations, the linear solver accounts for a significant (up to 55%) fraction of the overall runtime in virtually all FUN3D simulations.

Algorithm 1 SOLVER

```
Initialize q
for i ← 1 to maxiter do
    Construct Jacobian matrix A at q
    Construct vector b at q
    Solve linear equation AΔq = b for Δq
    q ← q + Δq
end for
```

In this paper, we describe a new mixed-precision linear-solver implementation that takes advantage of the IEEE 754 format for half-precision floating point (FP16) operations and further improves GPU performance. Traditionally, CFD solvers use single-precision (FP32) and double-precision (FP64) formats. In general, using lower precision in intensive computations improves performance by reducing the memory traffic or/and speeding up the execution of arithmetic operations by exploiting hardware support available for lower precision in the latest GPU architectures. Potential adverse effects of lower precision may include accuracy degradation and deterioration of iterative convergence. As the linear-solver kernel is memory bound on GPUs, we focus on reducing memory traffic by using the FP16 format while maintaining double-precision solution accuracy. A reduction in memory traffic directly translates to improved performance. To avoid adverse effects, the lower precision computations are used to compute corrections to a higher-precision solution and the lower-precision correction iterations restart frequently to prevent accumulation of round-off errors. The methodology is similar to the iterative refinement strategies of [6], [7]. The specific contributions of this paper include formulation, evaluation, and optimization of mixed-precision computations for large-scale CFD applications.

The paper is structured in the following manner. First, some details of the solver are presented. Recent advances in the implementation of the linear solver for GPUs are described. This double-single (DS) precision implementation uses FP32 and FP64 formats. A new mixed-precision linear solver is then introduced and analyzed; its double-single-half (DSH) precision algorithm uses the FP16 format for some critical computations. Accuracy and speed-up of the DSH GPU solutions are compared with the baseline DS GPU solutions, including an assessment for an unsteady turbulent-flow simulation on a grid with 1.14 billion grid vertices conducted on the Summit system at the Oak Ridge Leadership Computing Facility. The paper concludes with a brief summary and future research directions.

II. FUN3D SOLVER

A high-level description of the nonlinear solver representing iterations (Eqs. 2,3) for Eq. 1 is shown in Algorithm 1. The linearization matrix

$$\mathbf{A} = \frac{V}{\Delta\tau}\mathbf{I} + \frac{\partial\hat{\mathbf{R}}}{\partial\mathbf{q}} \tag{4}$$

and the nonlinear residual vector

$$\mathbf{b} = -\mathbf{R}(\mathbf{q}) \tag{5}$$

are computed at the current solution $\mathbf{q}$; $\mathbf{I}$ is the identity matrix. The correction, $\Delta\mathbf{q}$, is computed as an approximate solution of the linear system

$$\mathbf{A}\Delta\mathbf{q} = \mathbf{b} \tag{6}$$

For a spatial mesh containing n grid vertices, $\mathbf{A}$ represents a sparse $n \times n$ block matrix, where each matrix entry is a dense block of size $n_b \times n_b$ based on the linearization of the nonlinear governing equations at each grid vertex.

The implicit approach used within FUN3D requires frequent solutions of Eq. 6 during the course of a simulation. To optimize efficiency and memory usage, the matrix $\mathbf{A}$ is segregated into two separate matrices,

$$\mathbf{A} \equiv \mathbf{D} + \mathbf{O} \tag{7}$$

where $\mathbf{D}$ and $\mathbf{O}$ represent the diagonal and off-diagonal blocks of $\mathbf{A}$, respectively.

The blocks contained in $\mathbf{D}$ and $\mathbf{O}$ are stored separately. The block-sparse $n \times n$ matrix $\mathbf{O}$ contains nnz non-zero $n_b \times n_b$ blocks that are stored using a block compressed sparse row (CSR) format [4]. Each of the n rows and columns containing $n_b \times n_b$ blocks are referred to as a $brow$ and a $bcol$, respectively. Two integer arrays ia and ja are used to efficiently capture the sparsity pattern of the matrix. The array ia is a rank-1 array of size $n+1$ whose i-th entry indicates the leading non-zero block index in the i-th $brow$ of $\mathbf{O}$. The array includes a fictitious $n+1$ entry to facilitate traversal of the elements through the n-th $brow$. The ja array is a rank-1 array of size nnz that provides the $bcol$ index for each non-zero block. A third array $data$ is used to store the non-zero entries. Each $n_b \times n_b$ block is stored in column-major order. Several linear-solver options are provided within FUN3D; the scheme most commonly used in practice is a multicolor point-implicit relaxation. The algorithm to solve the linear system is shown in Algorithm 2. In this scheme, the grid vertices are grouped, or colored, such that no two adjacent (edge-connected) vertices are assigned the same color. Since the matrix $\mathbf{A}$ is constructed using only nearest-neighbor relations, unknowns defined at the grid vertices of the same color do not depend on each other and can be updated in parallel in a Jacobi-like fashion. Colors are processed sequentially. Updates of unknowns at grid vertices of each color use the latest updated values of $\Delta\mathbf{q}$ corresponding to other colors. The overall process may be repeated using several outer sweeps over the entire system. In a typical simulation, $n_{iter} = 15$ sweeps are performed within each nonlinear iteration. This number of linear iterations is empirically observed to result in suitable convergence of the nonlinear solver.

Algorithm 2 MULTICOLOR LINEAR SOLVER

1: **for** $i \leftarrow 1$ **to** n_{iter} **do**
2: **for** $c \leftarrow 1$ **to** n_c **do**
3: $\Delta \mathbf{r} \leftarrow \mathbf{b_c} - \mathbf{O_c} \Delta \mathbf{q}$
4: $\Delta \mathbf{q_c} \leftarrow \mathbf{D_c}^{-1} \Delta r$
5: **end for**
6: **end for**

To improve cache performance, the system of algebraic equations is renumbered such that unknowns corresponding to vertices of the same color appear in a consecutive order, and the arrays ia and ja are modified appropriately to reflect the new matrix structure. In Algorithm 2, $\mathbf{O_c}$ and $\mathbf{D_c}$ represent sub-matrices of $\mathbf{O}$ and $\mathbf{D}$ respectively defined by color c, and b_c represents the sub-vector defined by the color c, for $1 \leq c \leq n_c$. The value n_c represents the total number of colors, which is generally about a dozen for typical meshes encountered in practice. The vector $\Delta \mathbf{q}$ is initialized to zero. Line 3 of Algorithm 2 represents a standard block-sparse matrix-vector operation. Line 4 involves inversion of matrix $\mathbf{D_c}$. For efficient inversion, an lower-upper (LU) decomposition of $\mathbf{D_c}$ can be computed once and stored in place of $\mathbf{D_c}$. The solution for the current block row is then obtained through a forward-backward substitution procedure.

III. GPU IMPLEMENTATION OF LINEAR SOLVER

The GPU is best suited for computations that can be executed concurrently on multiple data elements. In general, a computation is partitioned into thousands of fine-grained operations, which are assigned to thousands of threads on a GPU device for parallel execution. The GPU hardware consists of a number of streaming multiprocessors (SMs), which in turn consist of multiple cores. Threads are organized in blocks, or cooperative thread arrays, where one or more blocks run on an SM. The threads in a block are further partitioned into subgroups of 32 threads known as warps. A warp runs on eight or sixteen cores of an SM in multiple clock cycles.

To develop an efficient GPU implementation of the multicolor point-implicit solver, functions provided by the *cuSPARSE* and *cuBLAS* [8] libraries were initially considered. The function *cusparseSbsrmv* multiplies a block-sparse matrix with a vector, and the function *cublasStrsmBatched* solves block systems of equations by performing forward and backward substitutions using an LU-decomposition matrix. Experiments showed that the performance of the library functions is suboptimal for linear systems representative of those encountered in CFD.

Optimized implementations of the *cusparseSbsrmv* and *cublasStrsmBatched* functions were proposed [5]. To perform a sparse matrix-vector product, the proposed algorithm allocates a number of warps, or a group of 32 threads, to process a subset of the blocks in one row of the sparse matrix. To perform forward and backward substitutions, a second kernel is invoked that assigns a single warp to process one diagonal block. Several challenges were encountered, including a variable extent of available parallelism, indirect memory addressing, low arithmetic intensity, and the need to accommodate different block sizes. To address these challenges, particular emphasis was placed on coalesced memory loads, the use of shared memory and pre-fetching, minimal thread divergence within warps, and strategic use of shuffle instructions available on recent hardware. Depending on block size, the new implementations show performance gains of up to 7x over the existing CUDA library functions.

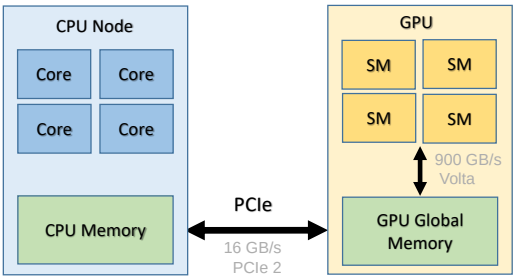

Fig. 1: CPU and GPU interface.

Although the solution of the linear system can be a significant fraction of overall runtime, all computational kernels required to advance the solution of the nonlinear governing equations should be moved to the GPU to realize maximum benefit of the architecture. A peripheral component interconnect express (PCIe) bus is used for moving data between the CPU and GPU. The bandwidth of the PCIe bus is an order of magnitude smaller than the memory bandwidth between the GPU main memory and SMs, see Figure 1. For example, PCIe 2 has a peak bandwidth of 16 GB/s, and the latest NVIDIA® Tesla V100 GPU has a peak bandwidth of 900 GB/s. In a typical application, the computationally intensive kernels are offloaded to the GPU. In this approach, all required data are moved to the GPU and results are brought back to the CPU once the GPU computation is completed. However, the time to move the data between the CPU and GPU can be significant compared to the GPU execution time. For this reason, we have moved the entire PDE solution procedure to the GPU as shown in Figure 2. In this paper, we focus on a mixed-precision implementation of the iterative linear solver. Future work will focus on mixed-precision matrix construction (Algorithm 1 line 3) for GPUs.

IV. MIXED-PRECISION ALGORITHM USING HALF-PRECISION FORMAT

In this section, we describe a new mixed-precision approach to further improve the performance of the linear solver on GPUs. Typically, the vectors $\mathbf{b}$ and $\mathbf{q}$ (Eq. 5) and the non-zero values of the block-diagonal matrix $\mathbf{D}$ (Eq. 7) are stored with double precision; the vector $\Delta \mathbf{q}$ (Eq. 6) and the non-zero values of the block-sparse matrix $\mathbf{O}$ (Eq. 7) are stored using

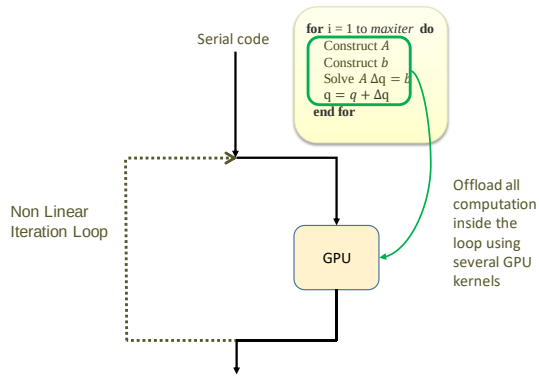

Fig. 2: Moving all main-loop computations to GPU.

single precision. This implementation using double and single precision is referenced as the DS implementation. To further decrease the amount of data to be stored and moved, the matrix $\mathbf{O}$ can be stored using half precision (16 bits). This implementation is referenced as the DSH implementation.

The IEEE 754 FP16 format requires 1 bit for the sign, 5 bits for the exponent, and 11 bits for the significand (10 bits are stored explicitly). The NVIDIA® Tesla P100 and V100 GPUs provide support for the FP16 format. The largest normal real number that can be represented using the FP16 format is 65504. Typical CFD applications may involve linearization matrices with entries that far exceed the largest FP16 real number. To avoid accuracy degradation, the non-zero values of the matrix $\mathbf{O}$ are scaled before converting them to half precision. The scaling ensures that the maximum converted value does not exceed 65504. The scaling factor β is given by:

$$\beta = 65504/MAXNZ,$$

where $MAXNZ$ is the maximum absolute non-zero value in the matrix $\mathbf{O}$. The mixed-precision linear solver for Eq. (6) is described in Algorithm 3.

Algorithm 3 MIXED PRECISION LINEAR SOLVER WITH FP16 FORMAT

1: $\mathbf{O^h} \leftarrow c2h(\beta\mathbf{O})$
2: **for** $i \leftarrow 1$ **to** n_{iter} **do**
3: **for** $c \leftarrow 1$ **to** n_c **do**
4: $\Delta\mathbf{r} \leftarrow \beta\mathbf{b_c} - \mathbf{O_c^h}\Delta\mathbf{q}$
5: $\Delta\mathbf{q_c} \leftarrow \beta^{-1}\mathbf{D_c^{-1}}\Delta r$
6: **end for**
7: **end for**

The operation `c2h` converts data stored in the FP32 format to the FP16 format. The DSH linear solver is implemented by altering slightly the existing CUDA DS linear solver implementation (Algorithm 2). Scaled entries of $\mathbf{O}$ are loaded as FP16 data and cast to the FP32 format to perform the matrix-vector product. The vector $\mathbf{b}$ is multiplied by the scaling factor β. The correction $\Delta\mathbf{q}$ is multiplied by $\frac{1}{\beta}$ prior to storage.

If we assume linear solver performance to be constrained entirely by memory bandwidth, then the speedup of the DSH implementation over the DS implementation is dictated by the amount of data loaded from main memory. This amount depends on n_b and the number of non-zero blocks in a row, $nnzr$. The speedup is computed as the number of bytes loaded in the DS implementation divided by the number of bytes loaded in the DSH implementation:

$$\frac{nnzr(n_b^2 \cdot s + n_b \cdot s + i) + d(n_b^2 + n_b) + n_b \cdot s + 2 \cdot i}{nnzr(n_b^2 \cdot h + n_b \cdot s + i) + d(n_b^2 + n_b) + n_b \cdot s + 2 \cdot i}$$

Here, h, s, and d denote the bytes required to represent a floating-point number with half, single, and double precision, respectively, and i denotes the bytes to represent an integer number. The number of non-zero blocks in a specific row depends on mesh connectivity and differs from row to row. For simplicity, we compute the speedup for $nnzr$ averaged over all rows of $\mathbf{A}$. In the limit $nnzr \to \infty$, the speedup approaches

$$\frac{n_b^2 \cdot s + n_b \cdot s + i}{n_b^2 \cdot h + n_b \cdot s + i}$$

and as $n_b \to \infty$, the speedup approaches $\frac{s}{h}$, which is 2 in this case.

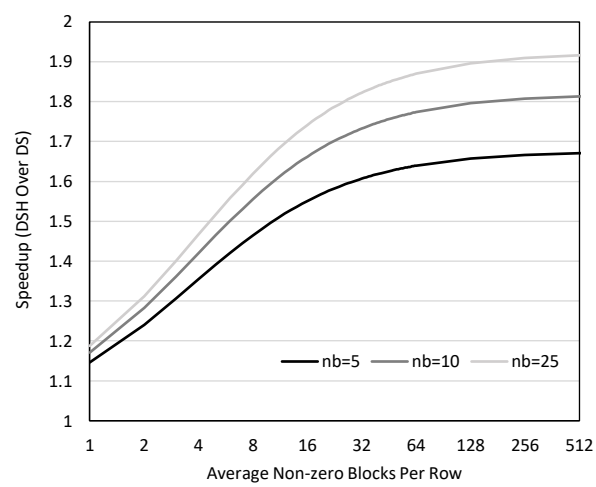

Fig. 3: Theoretical speedup for different block sizes and average numbers of non-zero blocks per matrix row.

Figure 3 plots theoretical speedup for different n_b and $nnzr$. In practical FUN3D simulations, the average value of $nnzr$ falls between 14 and 19, which is high enough to achieve over 90% of the theoretical speedup for a given n_b. In this paper, the value of n_b is fixed at 5. Improved speedup predicted for

higher values of n_b provides a strong motivation to pursue lower precision solvers for simulations involving reacting gas flows, which may require much larger values of n_b.

There are several ways to construct the $\mathbf{O^h}$ matrix in the FP16 format. In principle, the FP16 linearization matrix can be constructed directly; however, there are several significant obstacles to this approach. The CUDA kernels used to perform matrix construction rely on the atomic memory function, `atomicAdd`, to efficiently resolve race conditions. As of version 10.1.168, CUDA does not support 16-bit `atomicAdd`. The 32-bit function can be used, but memory traffic would not be reduced. Another obstacle is that the largest matrix entry and the sufficient scaling factor β are not known in advance. Matrix construction must be restarted with a reduced value of β if an exceedingly large matrix entry is encountered. With no clear benefits for constructing the FP16 matrix directly, we elect to compute the FP32 matrix $\mathbf{O}$ first and then to convert it to the FP16 matrix $\mathbf{O^h}$.

Because the matrix $\mathbf{O}$ accounts for approximately half of the application memory footprint, it must be converted to the FP16 format in place. Increasing memory usage by 25% could significantly degrade scaling performance since GPU memory is limited and scaling depends heavily on the ability to overlap communication with sufficient computation. Algorithm 4 describes a parallel in-place storage reduction algorithm (PIPSR). The specific PIPSR algorithm describes reducing the storage size by a factor of two; however, it can be readily adapted to reduce the original storage size by any integer factor.

If matrix $\mathbf{O}$ has an odd number of elements, the final element is processed separately. In the description below, it is assumed that the matrix $\mathbf{O}$ has even number of elements, n, and that the sufficient scaling factor β has been computed in advance. The algorithm uses three passes (**for** loops) over the memory space occupied by $\mathbf{O}$. The first pass multiplies each FP32 element i by β, converts it to the FP16 format, and stores the result in the first half of the space originally occupied by the FP32 element i. These actions leave the second half of the bytes of each original FP32 entry unused. In the second pass, the first $\frac{n}{2}$ FP16 entries are copied into the second half of the memory space, placed in the unused bytes created during the first pass. All FP16 entries now reside in the second half of the memory space, in the range from ($\mathbf{O^h_n}$ to $\mathbf{O^h_{2n-1}}$). The third pass copies FP16 entries from the second half of the memory space into the first half, restoring the correct order.

Each **for** loop of the algorithm has no loop-carried dependencies and is thus embarrassingly parallel. The loops must be completed sequentially. Note that the largest index of matrix $\mathbf{O^h}$ is twice as large as the largest index of matrix $\mathbf{O}$, but $\mathbf{O^h}$ and $\mathbf{O}$ still occupy the same memory space because the size of the elements of $\mathbf{O^h}$ is half the size of the elements of $\mathbf{O}$.

Algorithm 4 PARALLEL IN-PLACE STORAGE REDUCTION

```
if size(O) is even then
    n ← size(O)
else
    n ← size(O)−1
end if
for i ← 0 to n − 1 do                    # pass one
    O^h_2i ←c2h(βO_i)
end for
for i ← 0 to n/2 − 1 do                  # pass two
    O^h_(n+2i+1) ← O^h_2i
end for
for i ← 0 to n − 1 do                    # pass three
    if i < n/2 then
        O^h_i ← O^h_(n+2i+1)
    else
        O^h_i ← O^h_2i
    end if
end for
if size(O) is odd then                   # remainder loop
    O^h_n ← c2h(βO_n)
end if
```

The three passes of Algorithm 4 may be implemented using two CUDA kernels plus a remainder kernel. We fuse pass one and pass two into a single CUDA kernel. This is achieved by logically splitting the threads in a block into two, the first half of which load from the first half of $\mathbf{O}$, the second half load from the second half of $\mathbf{O}$, each group having the same offset. After reading, each block synchronizes. The memory location read by threads in the second half of the block may now be written by threads in the first half, indicated by pass two of Algorithm 4. Pass three is performed by the second CUDA kernel. Memory operations are coalesced by assigning chunks of the pass loops to warps of threads. This requires that the size of $\mathbf{O}$ be a multiple of the number of threads in a block. We process $\lfloor \frac{\text{size}(\mathbf{O})}{\text{threads\_per\_block}} \rfloor \times$ threads_per_block with the two kernels described and the remaining elements in a remainder loop, the cost of which is negligible for a typical $\mathbf{O}$.

On the NVIDIA® Tesla V100, the two-kernel-plus-remainder approach provides a speedup factor of 1.7 over a naive three-kernel-plus-remainder approach. Because the remainder loop allows us to choose arbitrarily the size of $\mathbf{O}$ processed by the main kernels, we can ensure all loads are aligned and thus use CUDA vector intrinsic types, increasing the efficiency of memory operations. Vector loads and stores increase the performance of PIPSR by a factor of 1.2. The bandwidth of the two kernels is measured at 846 and 826 GB/s, respectively, when n is sufficiently large, which is close to peak bandwidth. The two-kernel in-place approach is approximately twice as slow as simply copying $\mathbf{O}$ to $\mathbf{O^h}$ in a separate array.

V. RESULTS

The first test case is based on transonic turbulent flow over the semi-span wing-body configuration shown in Figure 4

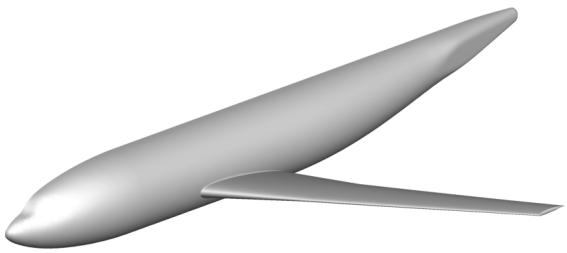

Fig. 4: Wing-body configuration taken from [9].

[9]. 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 1,123,718 grid vertices, 1,172,171 prisms, 3,039,656 tetrahedra, and 7,337 pyramids.

Iterative convergence for a linear system corresponding to this simulation is first examined. Here, a system $\mathbf{Ax} = \mathbf{b}$ has been extracted from the mean flow discretization of a single time step of the simulation. The linear residual history based on DSH and DS iterations is shown in Figure 5. The two plots are indistinguishable until iteration 35, after which the FP16 format cannot accurately represent updates of $\Delta\mathbf{q}$. Beyond this point, the residual convergence of DSH iterations stagnates as round-off error prevents further residual reduction while the residual of DS iterations continues to decrease. This result implies that DSH iterations should be frequently restarted; i.e., after several DSH iterations, the FP16 correction should be applied to correct the double-precision solution, the Jacobian matrix and nonlinear residual vector should be recomputed, and the half-precision correction should be re-initialized to zero. For the specific matrix considered here, the number of DSH iterations should not exceed 35. Recall that the typical number of linear solver iterations used in practice is 15.

A comparison of the DS and DSH approaches for the full steady-state turbulent flow simulation is shown in Figures 6 and 7. These simulations have been conducted using a single NVIDIA® Tesla V100 GPU on the Summit system at the Oak Ridge Leadership Computing Facility. The DSH methodology has been implemented for the meanflow linear system while the standard DS methodology is used for the one-equation turbulence model. Figure 6 shows the convergence of the root-mean-square (RMS) norm of the nonlinear density residual using both DS and DSH linear-solver implementations. There is no discernible difference in the convergence of nonlinear iterations using DS and DSH linear solvers. However, the DHS simulation is $1.20\times$ faster than the standard DS approach. This speedup is illustrated in Figure 7, which compares the wall-clock time of 10,000 nonlinear iterations using DS and DSH linear solvers; the wall-clock time to complete the simulation with the DS linear solver is taken to be 1.0. The plot breaks the simulation down into several components: *Linear_solver* is the subject of this paper; *LHS* is the formation of $\mathbf{A}$ of Eq. 4; *RHS* is the formation of the nonlinear residual vector $\mathbf{b}$ of Eq. 5; *Turb* is the turbulence model; *Misc* includes utility functions which transform data structures, compute norms, call CUDA API functions, etc.; *PIPSR* is Algorithm 4; *Overhead* includes everything not captured by the NVIDIA profile utility `nvprof`. The DSH linear solver is $1.53\times$ faster than the DS linear solver, however the speedup of the full simulation ($1.20\times$) is smaller because the linear solver constitutes only a fraction of the runtime. PIPSR takes $\approx 1\%$ of the runtime, and thus does not significantly impact the speedup. The scenario of a steady simulation on a single GPU leads to the greatest fraction of time used by the linear solver. From this perspective, the $1.20\times$ speedup observed in this scenario is likely the best achievable.

To evaluate the implementation of the DSH approach on a GPU architecture at scale, an unsteady Detached Eddy Simulation of supersonic retropropulsion is performed on 92 nodes of the Summit system (552 NVIDIA® Tesla V100 GPUs). This case uses a mesh containing 1,142,270,801 vertices, 263,608,400 prisms, 6,046,934,464 tetrahedra, and 283,836 pyramids. The freestream Mach number is 2.4, the angle of attack is zero degrees, and the Reynolds number based on the heat shield diameter is 5.89 million. Each of the eight nozzles operates at a specified pressure ratio of 8,733. An instantaneous snapshot of total temperature isosurfaces is shown in Figure 8.

Convergence of the meanflow density residual is shown in Figure 9 and performance breakdown is shown in Figure 10. As in the steady-simulation case, there is no discernible difference between the nonlinear residuals observed in the unsteady simulations using the DSH and DS linear solvers (figure 9). The overall speedup for the unsteady simulation

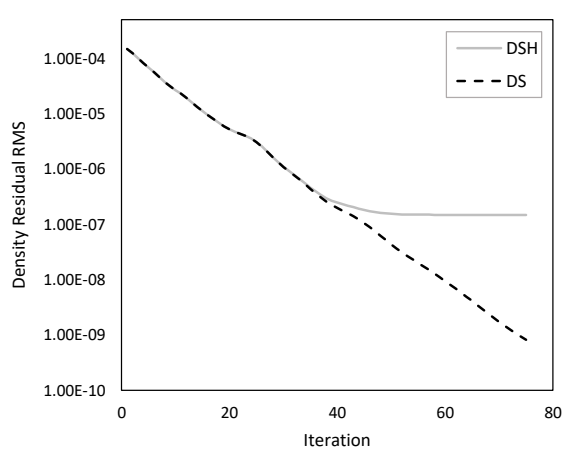

Fig. 5: DS and DHS iterations for a representative matrix consisting of 1,123,718 block rows.

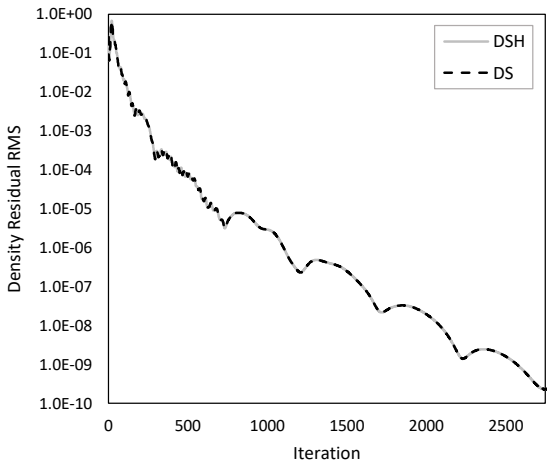

Fig. 6: Residual convergence history for steady RANS simulations on a single NVIDIA® Tesla V100 device.

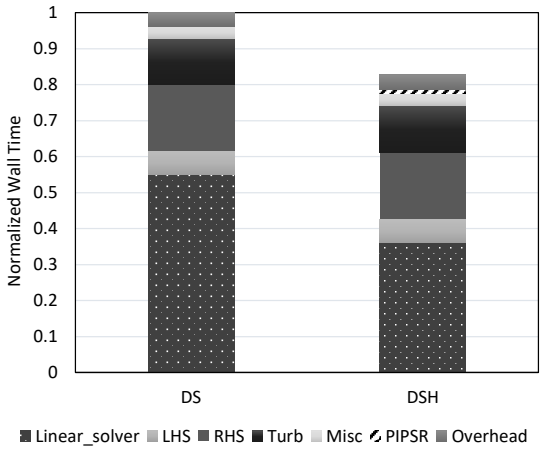

Fig. 7: Performance breakdown for steady RANS simulations on a single NVIDIA® Tesla V100 device.

using the DSH linear solver is $1.11\times$. There are several factors leading to the speedup that is lower than the speedup observed in the steady-simulation case. The linear solver constitutes a smaller portion of the unsteady computation. This can be seen by comparing the normalized wall time plots in Figures 7 and 10. A portion of the unsteady runtime is spent in MPI calls, which is reflected in the larger *Overhead* component. In addition, *LHS* is computed more frequently in the unsteady case. Finally, the DSH linear solver speedup is smaller in the unsteady simulation ($1.45\times$ instead of $1.55\times$) because of a lower average number of non-zero blocks per row in matrix $\mathbf{A}$.

VI. Summary and Future Work

A large-scale unstructured-grid computational fluid dynamics code, FUN3D, uses a linear solver to compute corrections to nonlinear solutions of the fluid dynamics equations. A double-single (DS) mixed-precision implementation of

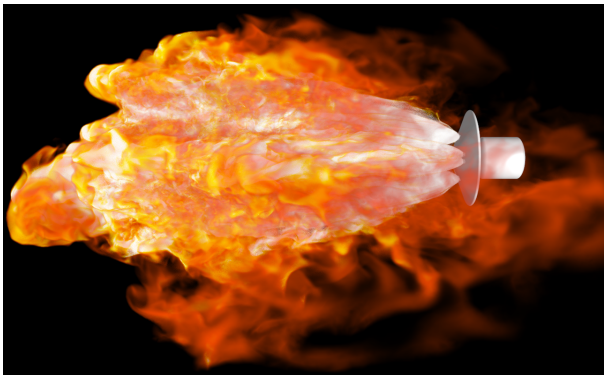

Fig. 8: Retropropulsion simulation performed on Summit. Freestream flow direction is from left to right.

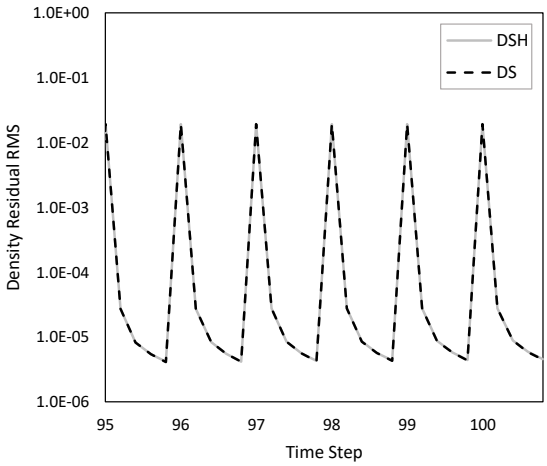

Fig. 9: Typical subiteration convergence for unsteady retropulsion simulations.

the linear-solver kernel has been previously optimized for performance on Graphics Processing Units (GPUs). In that implementation, the IEEE 754 double-precision floating-point (FP64) format is used to store the diagonal blocks of the Jacobian matrix and vectors of the nonlinear residuals and solutions; the single-precision (FP32) format is used to store the off-diagonal blocks of the Jacobian and correction vector.

This paper has presented a new mixed-precision implementation of the FUN3D linear-solver kernel to further improve GPU performance. The new implementation, referred to as the double-single-half (DSH) implementation, reduces memory traffic by using the half-precision (FP16) format while maintaining double-precision solution accuracy. In the new implementation, the off-diagonal blocks of the Jacobian are generated in the FP32 format, then scaled and stored in-place in the FP16 format. As the FUN3D linear-solver kernel is memory bound on GPUs, a reduction in memory traffic directly translates to improved performance. To maintain double-precision solution accuracy, lower-precision correction iterations restart frequently to prevent

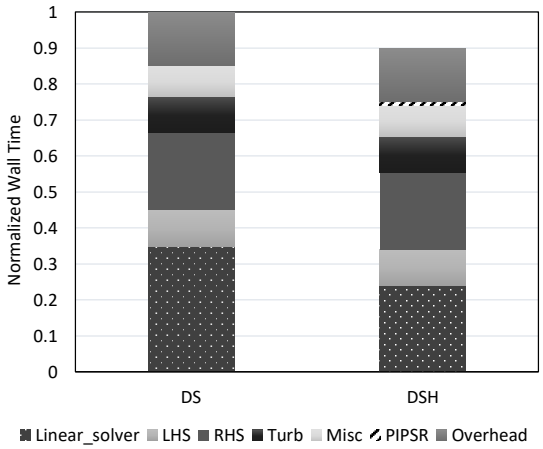

Fig. 10: Performance breakdown for unsteady retropropulsion simulations.

accumulation of round-off errors.

The performance of the new implementation has been assessed for a benchmark problem and a practical large-scale computation. The benchmark simulation computed a steady state for a turbulent flow on an unstructured grid with approximately one million grid vertices. The large-scale study performed a high-resolution simulation of an unsteady supersonic retropropulsion configuration on an unstructured grid with about 1.14 billion grid vertices. Both studies have been conducted using NVIDIA® Tesla V100 GPUs on the Summit system at the Oak Ridge Leadership Computing Facility. The solutions computed with the DS and DHS implementations are indistinguishable. The theoretical speedup of the DSH linear solver over the DS linear solver ($1.55\times$) is closely matched by the speedup ($1.53\times$) observed in the benchmark study. The total simulation time is reduced by a factor of 1.20 for the benchmark steady computation and 1.11 for the large-scale unsteady simulation.

In the future, we plan to study the benefits of lower-precision computations for FUN3D performance on different architectures, such as AMD GPUs and ARM processors. Mixed-precision GPU implementations of other iterative solvers and computational models implemented within FUN3D will also be explored.

Acknowledgments

We are grateful to the High Performance Computing Incubator at the NASA Langley Research Center for providing support for this work. The support of Dr. Mujeeb Malik, Technical Lead for the Revolutionary Computational Aerosciences sub-project within the NASA Aeronautics Research Mission Directorate Transformational Tools and Technologies Project, is also acknowledged. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] R. T. Biedron, J.-R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park, C. L. Rumsey, J. L. Thomas, and W. A. Wood, *FUN3D Manual 13.5*, NASA/TM-2019-220271, 2019.

[2] E. J. Nielsen and B. Diskin, "High-performance aerodynamic computations for aerospace applications," *Parallel Computing*, vol. 64, pp. 20 – 32, 2017.

[3] L. Wang, B. Diskin, R. T. Biedron, E. J. Nielsen, V. Sonneville, and O. A. Bauchau, "High-fidelity multidisciplinary design optimization methodology with application to rotor blades," *Journal of the American Helicopter Society*, vol. 64, no. 3, pp. 032 002:1 – 11, July 2019.

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

[5] M. Zubair, E. Nielsen, J. Luitjens, and D. Hammond, "An optimized multicolor point-implicit solver for unstructured grid applications on graphics processing units," in *Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18–25.

[6] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 47:1–47:11.

[7] H. Anzt, P. Luszczek, J. Dongarra, and V. Heuveline, "GPU-accelerated asynchronous error correction for mixed precision iterative refinement," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 908–919.

[8] NVIDIA. (2015) cuBLAS user guide. [Online]. Available: http://docs.nvidia.com/cuda/cublas/

[9] 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.