A Multi-Architecture Approach for Implicit Computational Fluid Dynamics on Unstructured Grids

Gabriel Nastac*, Aaron Walden†, Li Wang‡, and Eric Nielsen§
NASA Langley Research Center, Hampton, Virginia, 23681

Yi Liu¶
National Institute of Aerospace, Hampton, Virginia, 23666

Matthew Opgenorth‖
Sierra Space, Louisville, Colorado, 80027

Jason Orender**, Mohammad Zubair††
Old Dominion University, Norfolk, Virginia, 23529

High-performance computing (HPC) architectures are trending toward manycore paradigms such as graphics processing units (GPUs). Approximately half of the top 100 publicly disclosed supercomputers in the world utilize GPU accelerators for performance. This is in contrast to a decade ago, where there were only a few such machines in the top 100. It is not currently possible to compile and run legacy central processing unit (CPU) software efficiently on GPUs without significant refactoring. Though a number of frameworks offering performance portability exist, none offer a standardized specification that is supported by all major hardware vendors. Additionally, experiences show that obtaining a high percentage of peak performance often requires architecture-specific code. This work details a pragmatic multi-architecture computational fluid dynamics library focused on aerospace problems across the speed range from low subsonic to hypersonic flows involving thermochemical nonequilibrium. A thin abstraction layer above NVIDIA CUDA C++ is utilized, which enables primarily single-source software currently capable of running efficiently on multicore CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs. Results on various problems of interest across the speed range are presented and performance is compared between various architectures.

I. Introduction

A graphics processing unit (GPU) may be thought of as a specialized chip that uses over an order of magnitude greater concurrency than multicore central processing units (CPUs) to overcome latency and improve floating-point operation throughput. Since the advent of NVIDIA's Compute Unified Device Architecture (CUDA) [1] in 2007, which leverages GPUs for general-purpose computing, GPUs have dramatically accelerated a broad range of applications from deep learning [2] to traditional scientific simulations [3]. Current algorithms may require adaptation to exploit the concurrency and throughput of emerging hardware such as GPUs. Seven of the top ten supercomputers listed on the current TOP500 [4] utilize GPUs. Almost half of the current top 100 machines utilize NVIDIA GPUs. Emerging U.S. exascale systems rely on Intel and AMD GPU acceleration for performance [5–7]. Effective utilization of this next-generation hardware is key for design and analysis of next-generation aerospace vehicles according to the NASA CFD Vision 2030 study [8].

To effectively utilize manycore processors such as GPUs, algorithms and software must be appropriately adapted to exploit their high levels of parallelism. Many programming models for generating GPU code exist. One of the earliest

*Research Scientist, Member AIAA
†Research Scientist
‡Research Scientist, Associate Fellow AIAA
§Senior Research Scientist, Associate Fellow AIAA
¶Associate Principal Research Engineer, Senior Member AIAA
‖Principal Systems Engineer
**Graduate Researcher
††Professor

and most popular parallel programming models for GPUs is NVIDIA CUDA. CUDA is a C++-based programming model that enables users to execute compute kernels and control NVIDIA GPUs. One limitation of CUDA is that it is proprietary and only NVIDIA GPUs are supported. While NVIDIA GPUs are currently the most widely used GPUs for scientific computing, it is desirable to run the same software on multiple architectures. Open Computing Language (OpenCL) was initially developed by Apple and formally released by the Khronos Group in 2009. OpenCL is an independent standard for heterogeneous architectures including CPUs and GPUs [9]. The OpenCL C++ specification was introduced in OpenCL 2.1 in 2015. While open, some vendors have chosen not to support the latest standards. The latest OpenCL 3.0 standard requires only OpenCL 1.2 support; features beyond OpenCL 2.0 are optional. This optional support hinders the advancement of OpenCL as a portability standard. AMD's Heterogeneous Interface for Portability (HIP) is an open source software stack for GPU computing released in 2016 [10]. HIP is a C++ dialect, an abstraction similar to CUDA, designed to enable portable GPU C++ code for AMD and NVIDIA GPUs. AMD provides tools to convert CUDA to HIP. AMD is also working on a HIP CPU runtime to enable CPUs to run HIP code. CHIP-SPV [11] is a HIP implementation in development cosponsored by Argonne National Laboratory that supports OpenCL and Intel Level Zero [12] to support upcoming Intel GPUs. Directive-based approaches include OpenMP [13] and OpenACC [14]. OpenMP has supported GPUs since version 4.0 in 2013. SYCL is a cross-platform abstraction layer standardized by the Khronos Group that allows users to write single-source compute kernels in ISO C++. Initially released in 2014 [15], SYCL 2020 generalized the standard to enable the use of any API such as PTX for NVIDIA GPUs. Multiple implementations exist including Intel Data Parallel C++ (DPC++), released in 2020 [16], and ComputeCpp by Codeplay [17], a software company now owned by Intel. Other vendors have yet to fully support the evolving standard. Kokkos is a C++ programming model developed primarily by Sandia National Laboratory and released in 2014 [18] to enable performance portable applications for HPC platforms. RAJA is a similar portability layer developed at Lawrence Livermore National Laboratory also released in 2014 [19]. There is also an effort to extend ISO C++ to support platform-agnostic parallelism [20]. There are many different approaches to accelerator programming and this list is not exhaustive (e.g., Refs. [21, 22]). Ideally, programmers would write software in a high-level language with standardized constructs supported by major hardware vendors and obtain satisfactory performance across the spectrum of contemporary HPC architectures.

Although many early adopters of HPC architectures based on NVIDIA hardware leveraged the popular CUDA ecosystem, next-generation leadership-class supercomputers currently being procured by the Department of Energy (DOE) rely on AMD and Intel GPU hardware. The DOE Exascale Computing Project (ECP) details 22 different programming models used across their 62 application codes, with CUDA, Kokkos, HIP, and OpenMP used in 24%, 23%, 19%, and 19% of codes, respectively [23]. The ECP has not yet reached a clear consensus with regard to GPU programming models.

The NASA Langley Research Center unstructured-grid CFD solver FUN3D [24], originally a Fortran code, has recently been adapted for NVIDIA GPU architectures [25, 26] and has been used to demonstrate excellent performance for perfect gas and thermochemical nonequilibrium flows [27]. Recent work on the Oak Ridge National Laboratory Summit system [28] has demonstrated the ability of FUN3D to run efficiently on thousands of NVIDIA V100 GPUs [29, 30]. This was done by rewriting the traditional Fortran kernels in CUDA C++. At the time, it was decided by the FUN3D developers that CUDA would provide the most acceptable balance between performance and productivity [25]. Performance is paramount for GPU adoption by research organizations such as NASA, which often use legacy software. Not only must GPUs deliver higher performance per dollar than CPUs, they must also justify the cost of rearchitecting legacy software.

To run FUN3D on all major CPU and GPU architectures, a thin abstraction layer over FUN3D's CUDA code has been developed. The layer abstracts GPU concepts such as kernel launching and shared memory to enable a single source file to be compiled by different vendor compilers to produce code for different architectures. This enables FUN3D to run on multicore CPUs through ISO C++, NVIDIA GPUs through CUDA C++, AMD GPUs through HIP, and Intel GPUs through SYCL. These abstractions are used in the FUN3D Library for Universal Device Acceleration, henceforth FLUDA. FLUDA was originally developed for NVIDIA GPUs, thus the code is structured differently than the original Fortran. To run FLUDA on CPUs, this GPU-oriented code is run with a single thread (with MPI for parallelism). It is our hypothesis that modern, superscalar CPUs will better tolerate GPU-oriented code than the reverse. To ensure high performance, several key kernels in FLUDA have architecture-specific implementations [31–33]. This architecture-specific code comprises ≈2% of the library.

This paper will first describe the governing equations and numerical methods for the flow solver. The abstractions will then be described including the design and implementation of the kernels. Next, results will be demonstrated including device-level performance using the proposed approach on various problems across the speed range.

II. Governing Equations and Numerical Methods

FUN3D, an unstructured-grid node-based finite-volume CFD solver developed and maintained at the NASA Langley Research Center, is widely used for analysis of turbulent flows across the speed range from incompressible to hypersonic regimes [24]. Specifically, we are focusing on the perfect gas and thermochemical nonequilibrium capabilities of the code for this work, although an incompressible gas model is also available and supported by the current approach. The governing equations for compressible reacting flow are the conservation of species, mixture momentum, and total energy:

$$\frac{\partial}{\partial t}\left(\rho y_s\right)+\frac{\partial}{\partial x_j}\left(\rho y_s u_j\right)-\frac{\partial}{\partial x_j}\left(J_{sj}\right)=\dot{\omega}_s, \tag{1}$$

$$\frac{\partial}{\partial t}\left(\rho u_i\right)+\frac{\partial}{\partial x_j}\left(\rho u_i u_j+p\delta_{ij}\right)-\frac{\partial}{\partial x_j}\left(\tau_{ij}\right)=0, \tag{2}$$

$$\frac{\partial}{\partial t}\left(\rho E\right)+\frac{\partial}{\partial x_j}\left(\left(\rho E+p\right)u_j\right)-\frac{\partial}{\partial x_j}\left(u_k\tau_{kj}+\dot{q}_j+\sum_{s=1}^{N_s}h_s J_{sj}\right)=0, \tag{3}$$

where y_s is the mass fraction of species s, $\rho=\sum_{s=1}^{N_s}\rho_s$ is the mixture density, $\rho_s=\rho y_s$ is the species density of species s, u_i is the ith component of the velocity, and E is the total energy. J_{sj} is the jth component of the diffusive flux, $\dot{\omega}_s$ is the chemical source term of species s, p is the pressure, τ_{ij} is the shear-stress tensor, $\dot{q}_j$ is the jth component of the heat flux, and h_s is the enthalpy of species s. Pressure is modeled as an ideal gas, heat transfer is modeled with Fourier's law, species diffusion is modeled using Fick's law, and the shear-stress is modeled with a Newtonian model. NASA polynomials [34] are used to compute thermodynamic properties on a per-species basis. The transport properties (diffusivity, viscosity, and thermal conductivity) are computed using collision integrals [35]. The compressible perfect gas equations can be obtained by assuming a single perfect gas species with thermally perfect thermodynamic properties and simplified transport properties (e.g., constant Prandtl number and Sutherland's law). The full details of the governing equations, including the two-temperature model for high-temperature flows [36], can be found in Ref. [35].

A median-dual finite-volume approach is used for spatial discretization on general unstructured grids containing tetrahedra, pyramids, prisms, and hexahedra. A dual-grid is generated by bisecting every edge and connecting them to cell centroids to generate polyhedra. Inviscid fluxes are computed at each dual-face using an approximate Riemann solver with second-order accuracy obtained using an unstructured Monotonic Upstream-centered Scheme for Conservation Laws (UMUSCL) reconstruction [37] with unweighted least-squares gradients. Viscous fluxes are computed using a novel edge-based method [38], equivalent to a finite-element Galerkin-based approach. The discrete equations are generally integrated in time using implicit backward-difference formulae, although explicit time integration is also supported. For thermochemical nonequilibrium flows, chemistry is solved fully coupled with momentum and energy, with the option to tightly couple turbulence or solve loosely using the same approach and implementation used for perfect and incompressible gas applications. A multicolor point-implicit approach is used to obtain approximate solutions of the linear system of equations at each nonlinear iteration using 15 sweeps by default.

III. Software Design and Implementation

FUN3D was originally a CPU-based code written in Fortran. Coarse-grained message passing via the MPI standard [39] with conventional domain decomposition techniques are used for parallelism, although a hybrid MPI+OpenMP [13] implementation is also partially supported. Viewed from a high level, FUN3D is organized as a series of compute kernels that transform mutable data structures (examples include the residual vector and Jacobian matrix). The original development of the GPU-only FLUDA is detailed in Ref. [27]. FLUDA contains all compute kernels necessary for solving the discretized governing equations. FLUDA currently comprises roughly 60,000 lines of code and includes support for incompressible, perfect gas, and thermochemical nonequilibrium gas flows with multiple turbulence models.

Recently, FUN3D has been refactored to facilitate component-based development [40]. FUN3D has been rewritten as a high-level C++ driver, tentatively named NCFV for Node-Centered Finite-Volume, which uses the same FLUDA kernels as FUN3D's legacy Fortran-based flow solver. NCFV has been designed as an architecture-aware host plus device(s) model as its first principle. NCFV utilizes an interoperable API [41] to facilitate sharing of related functionality (e.g., I/O, grid, visualization) with other FUN3D components such as the high-order stabilized finite element solver (SFE) [42]. This section will detail the multi-architecture GPU-first design approach.

A. GPU-First Approach to Performance Portability

Legacy scientific software begins as a CPU code as a matter of necessity, generally speaking. FUN3D was first a Fortran code. FUN3D's GPU library, FLUDA, was adapted from that Fortran code into CUDA C++. This is presumably the way the majority of scientific codes evolve from CPU-only to GPU-accelerated (though not necessarily using CUDA as the GPU programming model). AMD GPU support was added to FLUDA using a thin abstraction layer that translates to HIP or CUDA through C preprocessor macros. It soon became clear that CPU support could be added through these macros by treating the CPU as equivalent to a GPU with a single thread. In FUN3D's case of MPI-only parallelism, this equivalence holds generally. Extension to SYCL was performed with the same abstractions enabling Intel GPU support. In total, the preprocessing macros comprise roughly 500 lines of code. While it is possible in theory to run on many architectures with some of the programming models we employ here, it has not yet been attempted and is left for future work. For example, HIP can be used on NVIDIA and AMD GPUs, CPUs (although HIP-CPU development is still a work in progress), and Intel GPUs (also in progress). SYCL can also be used on these architectures.

Listing 1 shows a generalized example FLUDA kernel that loops over the edges of a grid. Abstraction macros are typically capitalized versions of CUDA constructs such as `atomicAdd`. These macros will be translated appropriately for each architecture via the header file `platform.h`. In the case of constructs such as `THREADIDX_X`, the CPU is treated as a GPU with a single thread. Note that use of a grid-stride loop may not always be possible or may hamper performance. In these scenarios, a `CPU_GPU` macro is used, which becomes its first argument if compiling for CPU and its second otherwise. This macro is most commonly used for loop control. It is possible to use a grid-stride loop in place of the `CPU_GPU` macro to avoid code divergence, as is done in Listing 1, but experience has shown that GPU compilers will use fewer registers per thread when an `if` condition is used in place of a grid-stride loop. We find that, in FLUDA, register pressure is the strongest constraint on GPU performance.

Listing 1 Example FLUDA edge (dual-face) kernel.

```
#include "platform.h"

#define EDGE_CONNECTIVITY(j,i) edge_connectivity[ (j) + (size_t)(i)*2 ]

__DEVICE__
double f(const double varl, const double varr)
{
  return 0.5*(varl+varr);
}

__GLOBAL__
void example_kernel(const int nedges, const int* edge_connectivity, const double* var,
    const double* area, double* residual)
{
    int n = BLOCKIDX_X * BLOCKDIM_X + THREADIDX_X;
    int grid_size = BLOCKDIM_X * GRIDDIM_X;

    // CPU_GPU(for(;n < nedges; n++), if (n < nedges))
    for(;n < nedges; n += grid_size)
    {
     int nodel = EDGE_CONNECTIVITY(0,n); // Left state
     int noder = EDGE_CONNECTIVITY(1,n); // Right state

     double flux = f(var[nodel],var[noder]) * area[n];

     ATOMICADD(&residual[nodel],  flux);
     ATOMICADD(&residual[noder], -flux);
   }
}
```

The interior inviscid Jacobians are used as an example to emphasize the impact of a GPU-first approach. A 5-species, two-temperature model physics set is used, which leads to Jacobian blocks of size 10 by 10. A loop over edges (dual-faces) of the grid is performed, and the Jacobians are computed and added to the global data structures to form the global Jacobian matrix. The original Fortran implementation computes and stores derived state at each edge as well as the intermediate Jacobians themselves before finally adding the contributions to the global data structures. This kernel is naïvely converted to FLUDA in Listing 2. The primary difference between the Fortran and FLUDA naïve implementation is that variable length arrays (VLAs) are not standard in C++, and therefore, in FLUDA, a maximum bound is used on species, energies, and turbulence variables (11, 2, and 2, respectively) to size the local data. FLUDA kernels are generally templated on the specific physics set for performance reasons, but templates were not used in the naïve example to match the Fortran as closely as possible. This kernel can compile without modifications and run on CPUs through ISO C++, NVIDIA GPUs through CUDA, AMD GPUs through HIP, and Intel GPUs through SYCL using FLUDA C preprocessor macros. The naïve FLUDA CPU kernel is roughly 1.35× faster than the Fortran kernel, attributed primarily to the compile-time local array sizes mentioned previously. However, running this naïve kernel will lead to poor performance on GPU architectures. Storing a Jacobian in local memory on each thread will greatly increase register pressure, which lowers GPU occupancy and can lead to spilling, which may require expensive global memory operations. Table 1 shows relative performance for the Fortran and FLUDA kernels running a grid consisting of approximately one million points. A dual-socket Intel Skylake 6148 CPU node (40 cores) outperforms the naïve parallel kernel on an NVIDIA V100 GPU by a factor of two. We expect a factor of ≈3.5 in favor of the NVIDIA V100 GPU based on the hardware memory bandwidth ratio, as FUN3D kernels have low arithmetic intensity and are thus bound by main memory performance.

Employing hierarchical parallelism, as shown in Listing 3, reduces register pressure and can improve occupancy and performance. In this approach, a group of threads works on each edge. The edge data can be stored in shared memory instead of thread-local memory. Each component of the Jacobians is computed in place and added to the global data in a coalesced fashion. This hierarchical kernel runs on all architectures without device-specific code. The hierarchical kernel runs 1.12× slower than the naïve kernel on the Skylake CPU (though still faster than Fortran), but is 7.60× faster than the naïve kernel on the V100 GPU. The relative performance between the optimal CPU and optimal GPU kernels is roughly the memory bandwidth ratio.

Listing 2 Naïve parallel edge (dual-face) Jacobian kernel.

```
template<class CONFIG>
__GLOBAL__
void example_kernel_edge_parallelism(...)
{
  constexpr int N = CONFIG::N;
  // local variable declarations

  int n = BLOCKIDX_X*BLOCKDIM_X+THREADIDX_X;
  int grid_size = BLOCKDIM_X*GRIDDIM_X;

  for(;n < nedges; n += grid_size)
  {
    // Compute derived state and store

    // Compute and store Jacobians (NxN)
    for (int k=0;k<N;k++) {
        for (int j=0;j<N;j++) {
            // Compute (j,k) components
        }
    }

    // Add Jacobians to global data
  }
}
```

Listing 3 Hierarchical parallel edge (dual-face) Jacobian kernel.

```
template<class CONFIG>
__GLOBAL__
void example_kernel_hierarchical_parallelism(...)
{
  constexpr int N = CONFIG::N;
  // local and shared variable declarations

  int n = BLOCKIDX_X*BLOCKDIM_Y+THREADIDX_Y;
  int grid_size = BLOCKDIM_Y*GRIDDIM_X;

  for(;n < nedges; n += grid_size)
  {
    if (THREADIDX_X == 0) {
      // Compute minimum derived state
      // and store in shared memory
    }

    __SYNCTHREADS();

    for (int i=THREADIDX_X;i<N*N;i+=BLOCK_DIM_X)
    {
      int j = i / N;
      int k = i % N;

      // construct (j,k) components in place
      // and add to global data
    }
  }
}
```

Table 1 Relative performance of naïve and hierarchical parallel implementations of the inviscid Jacobian for 5 species and a two-temperature model (10 equations). Speedup is device-normalized with respect to FLUDA naïve Intel Skylake 6148. The final column shows vendor-reported maximum hardware memory bandwidth (MBW) normalized to Intel Skylake 6148.

Architecture	Implementation	Speedup	Hardware MBW Ratio
Intel Skylake 6148	Fortran	0.73	1.00
Intel Skylake 6148	FLUDA Naïve	1.00	1.00
Intel Skylake 6148	FLUDA Hierarchical	0.89	1.00
NVIDIA 16 GB SXM V100	FLUDA Naïve	0.58	3.52
NVIDIA 16 GB SXM V100	FLUDA Hierarchical	4.41	3.52

B. Kernel Optimizations

FUN3D kernels of significance typically undergo heavy refactoring from Fortran to CUDA to achieve optimal GPU performance [27]. Basic optimization strategies such as memory coalescing [43] are employed. Automatic (thread-local) arrays, used heavily in FUN3D, can significantly hinder GPU performance if not minimized. These arrays are minimized through parallelization, recomputation, and use of shared memory, though this is not an exhaustive list of strategies. Hierarchical parallelism is used when a kernel contains multiple independent intermediate computations within a work item. If there is significant thread divergence, multiple warps (or equivalent) are used. Aggressive use of C++ templates to exclude unnecessary branches and define values at compile time can significantly reduce GPU register pressure and thread divergence. Warp aggregation and other advanced strategies are discussed in Ref. [44]. Finally, GPUs are in-order processors and may be sensitive to the ordering of code statements. Through trial-and-error, the code is reorganized heuristically until performance improves. After each optimization, the threading is tuned (see Sec. III.B.1).

FUN3D optimizations are focused on kernels that dominate the run time. FUN3D is top-heavy in terms of kernel performance: ten kernels (out of roughly fifty per simulation) are responsible for 90% of the run time. The linear solver kernel is at least half the run time for most problems of interest. The top 3 kernels comprising 75% of the total run time are all memory-bound and reach significant percentages of peak memory bandwidth (MBW) reported by NVIDIA's profiler and appear in Table 2. The memory performance of the linear solver has been investigated in detail in past studies and has been confirmed with manual calculations [31–33].

Table 2 Percentage of peak memory bandwidth for the top 3 memory-bound kernels comprising 75% of the total run time on an NVIDIA 40 GB SXM A100 GPU for a 3.7M point mixed-element grid.

Kernel	Peak Memory Bandwidth [%]
Linear Solver	78
Viscous Jacobians	54
Inviscid Jacobians	62

1. Autotuner

FLUDA contains an autotuning framework that chooses optimal thread block parameters for a given kernel, template parameters, and GPU architecture. FLUDA GPU kernels are typically launched with two-dimensional thread blocks. The Y-thread index controls the number of work items (edges, in this case) processed by a block and the X-thread index controls hierarchical parallelism within the work item. A matrix construction loop might prepare a 5×5 matrix in shared memory using a single X thread and enlist the remaining X threads to write the matrix in a coalesced fashion to global memory. When enabled, the autotuner calls the kernel with many different combinations of X and Y threads and records the best time. After the fastest thread block size is found, the launch bounds will also be tuned, which can further lower register use and improve GPU occupancy.

2. Architecture-Specific Code

GPU optimizations as described in Sec. III.B often improve FLUDA kernel performance on CPUs relative to the legacy Fortran version (see Sec. IV). There are, however, instances in which architecture-specific code is required to achieve optimal performance. This is true for FUN3D's most expensive kernel, the linear solver. For this routine, the structure required for optimal GPU performance does not make effective use of the CPUs large resources per thread or its vector units. For this case, the `CPU_GPU` macro is used to call a specialized CPU function. Overall, there is less than 2% code divergence between architectures in FLUDA.

IV. Results

The first case examined is a RANS simulation of the NASA Common Research Model (CRM) at a transonic condition run to machine zero iterative convergence on Intel and AMD CPUs and NVIDIA, AMD, and Intel GPUs. This case demonstrates correctness on all architectures and is used to benchmark selected architectures. The second case is an unsteady wall-modeled large-eddy simulation (WMLES) of the NASA CRM at takeoff and landing at low Mach number conditions. This example demonstrates the approach for unsteady scale-resolving turbulent flows using many CPUs and GPUs. A rotor in hover is then used to demonstrate moving bodies. Lastly, two hypersonic cases demonstrating thermochemical nonequilibrium are examined. First, the Crew Exploration Vehicle (CEV) capsule is investigated at a high-enthalpy hypersonic wind tunnel experimental condition. Second, the Sierra Space Dream Chaser© spaceplane is investigated as a more complex configuration representative of next-generation aerospace vehicles.

The results presented are the fastest times obtained after testing a variety of compilers and optimization flags. Fortran results were obtained using Intel `ifort` 19.0.5.281. FLUDA CPU results are obtained using `g++` 10.3. GPU results are obtained using vendor-specific compilers. HPE MPT is used on NASA systems and OpenMPI is used on other systems. GPU-aware MPI is not used for the current results.

A. RANS Simulation of the NASA CRM at Transonic Conditions

The NASA Common Research Model (CRM) is used to demonstrate device-level performance for architectures of interest [45]. Specifically, the Drag Prediction Workshop 4 test case 1 is analyzed. The geometry and CFD grid are shown in Fig. 1. The mixed-element grid contains 3,675,916 vertices and 9,794,089 elements and is generated from a workshop tetrahedral grid[1] using FUN3D's *vgrid_merge_into_prisms* utility. The flow conditions are $M_\infty = 0.85$, $Re_C = 5 \times 10^6$, $C_L = 0.5$, and $T_\infty = 560$ °R; an angle of attack of 2.41° is used to match the target lift coefficient. The SA turbulence model is used [46] with a quadratic constitutive relation [47] (SA-QCR2000). Compact nearest-neighbor edge-based viscous terms are employed [48]. Local time stepping is utilized with maximum meanflow and turbulent Courant-Friedrichs-Lewy (CFL) numbers of 200 and 30, respectively. In total, 3000 iterations are run, with nonlinear convergence obtained at roughly 2500 iterations.

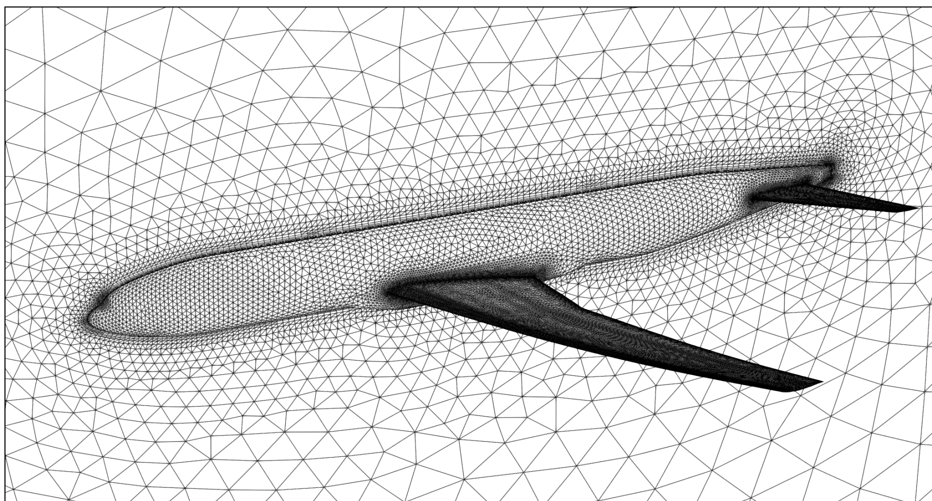


Fig. 1 NASA CRM aircraft grid.

[1] `https://dpw.larc.nasa.gov/DPW4/unstructured_Larc/NodeBase/dpw_wbt0_crs-3.6Mn_5.tgz`; Last Accessed May 2, 2022.

Various architectures are examined using FLUDA: a dual-socket Intel Skylake 6148 CPU node (40 total cores), a dual-socket AMD EPYC 7742 CPU node (128 total cores), an NVIDIA 16 GB SXM V100 GPU (the fastest of common variants), an NVIDIA 40 GB SXM A100 GPU (which has a lower memory bandwidth than the 80 GB variant), an AMD MI210 GPU, and an Intel Data Center GPU Max Series (previously code named Ponte Vecchio). The legacy Fortran implementation is run on the CPU nodes for comparison. Final drag values are listed in Table 3. All coefficients agree to machine precision. Density residual convergence is plotted for all architectures in Fig. 2. Residual convergence is also identical across architectures.

Device performance is presented in Table 4. Time to solution in seconds is presented. Performance generally correlates with memory bandwidth. Vendor-reported maximum memory bandwidth is theoretical and not necessarily obtained in practice [32]. Of note is that the FLUDA CPU implementation is faster than the Fortran CPU implementation. The Fortran CPU code used in this case has been optimized and refactored with substantial vendor support to achieve a speedup of $\approx 6.5\times$ [49], and thus can be considered an optimized basis. The fastest reported device time is 7 minutes for the NVIDIA 40 GB SXM A100 GPU. The kernel breakdown for a full time step (including the residual, Jacobian, and linear solve portions) for this device is plotted in Fig. 3. For a full time step, the linear solver is approximately half the run time and the residual and Jacobian each approximately one quarter of the run time. For this example, the Jacobian is updated in only 13% of the iterations, shifting the run time to the linear solver ($\approx$ 66% of run time) and residual computations. NVIDIA GPU simulations with more Jacobian evaluations will see improved relative performance due to high optimization levels for these complex kernels on NVIDIA architectures.

Table 3 Final CRM drag coefficient values for the various architectures.

Architecture	Implementation	C_D
Intel Skylake 6148	Fortran	1.435840192×10^{-2}
Intel Skylake 6148	FLUDA	1.435840192×10^{-2}
AMD EPYC 7742	Fortran	1.435840192×10^{-2}
AMD EPYC 7742	FLUDA	1.435840192×10^{-2}
NVIDIA V100	FLUDA	1.435840192×10^{-2}
NVIDIA A100	FLUDA	1.435840192×10^{-2}
AMD MI210	FLUDA	1.435840192×10^{-2}
Intel GPU	FLUDA	1.435840192×10^{-2}

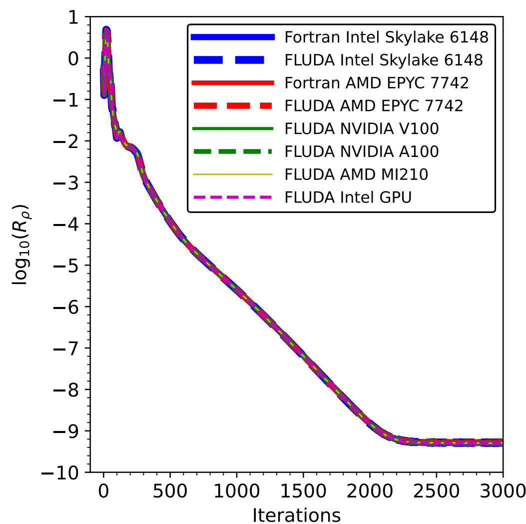


Fig. 2 Density residuals for various architectures.

Table 4 Time to solution for various architectures. Speedup is device-normalized with respect to FLUDA Intel Skylake 6148. Vendor-reported maximum hardware memory bandwidth (MBW) is normalized with respect to Intel Skylake 6148.

Architecture	Implementation	Time [s]	Speedup	Hardware MBW Ratio	Hardware MBW [GB/s]
Intel Skylake 6148	Fortran	3365	0.84	1.00	256 [50]
Intel Skylake 6148	FLUDA	2818	1.00	1.00	256
AMD EPYC 7742	Fortran	1738	1.62	1.60	409.6 [51]
AMD EPYC 7742	FLUDA	1639	1.72	1.60	409.6
NVIDIA 16 GB SXM V100	FLUDA	720	3.91	3.52	900 [52]
NVIDIA 40 GB SXM A100	FLUDA	430	6.55	6.05	1555 [53]
AMD MI210	FLUDA	620	4.55	6.40	1638.4 [54]

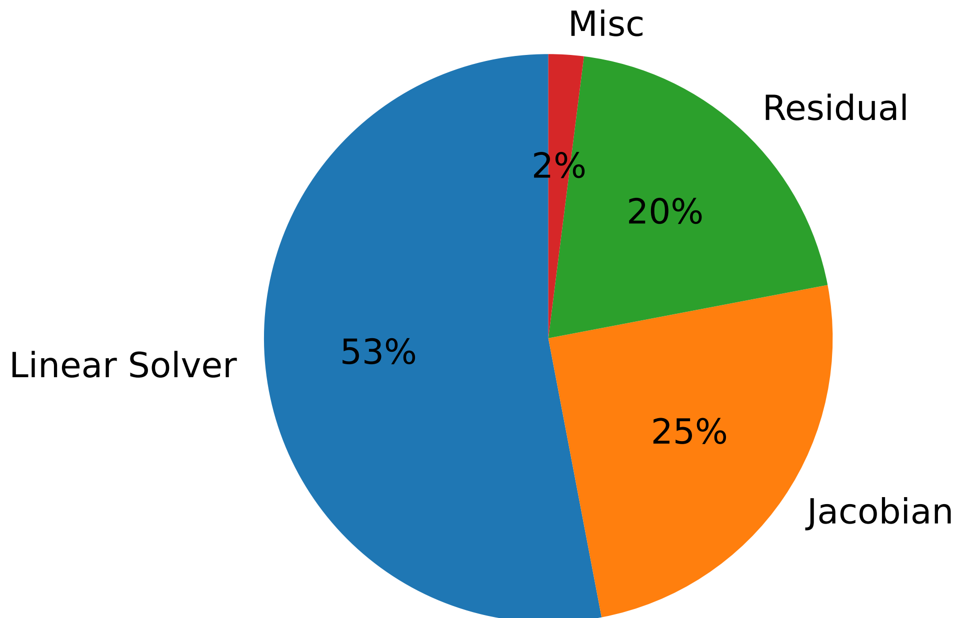


Fig. 3 Kernel breakdown for a full time step for NASA CRM on an NVIDIA 40 GB SXM A100.

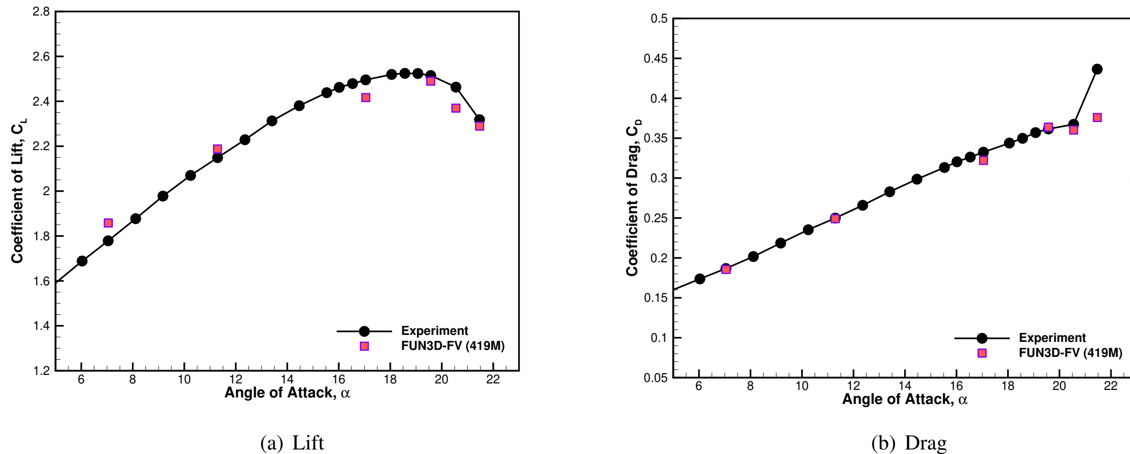
B. WMLES of the NASA High-Lift CRM

A WMLES capability has recently been implemented in FUN3D. The WMLES implementation largely follows the methodology described in Ref. [55]. At each time step, flow field variables are extracted at predetermined exchange locations, which are the first points off the wall in the present work. The extracted solutions serve as input to a simple equilibrium wall model [56] to determine the local shear stress at each grid point on the solid-wall surface. Once the shear stress is determined, it is used to compute boundary-face fluxes required for evaluating the residuals at the boundary grid points. More details about the WMLES implementation in FUN3D can be found in Refs. [57, 58]. The WMLES capability was initially enabled and assessed with CPU-centric legacy kernels and has been implemented in FLUDA to facilitate computations in a multi-architecture environment.

In this section, WMLES solutions and computational efficiency are evaluated for the NASA High-Lift Common Research Model (CRM-HL) at a range of angles of attack with the flaps in a landing configuration. The NASA CRM-HL configuration is a 10%-scale, semispan model that was tested in the QinetiQ 5-meter pressurized low-speed wind tunnel [59]. The experimental data are used for evaluating CFD simulations as part of the 4th AIAA CFD High Lift Prediction Workshop [60]. Free-air simulations are conducted for the nominal trailing-edge flap deflections of 40°/37°. The flow conditions for this configuration are a freestream Mach number of 0.2, a Reynolds number of 5.49 million based on the mean aerodynamic chord (MAC) of 275.8 inches and a reference temperature of 521 °R. A range of angles of attack have been considered in this study including 7.05°, 11.29°, 17.05°, 19.57°, 20.55°, and 21.47° for $C_{L,max}$ predictions and to compare with wall-corrected experimental data. More details about the geometry, test parameters, and related references are available from the workshop website [60].

The computational grid used in this work consists of approximately 419 million points and 812 million mixed-type elements including hexahedra, tetrahedra, prisms, and pyramids. All surface boundaries for the fuselage, slat, wing, flap, and nacelle use the wall model boundary condition. The nominal wall-normal spacing for this grid is approximately 0.13 inches, corresponding to a nominal $\Delta y_w^+ \approx 100$ based on a flat-plate approximation at freestream conditions and the MAC reference length. The symmetry plane is located at $y = 0$. The outer boundary is located approximately 100 MAC lengths away and uses a farfield Riemann invariant boundary condition. The simulations have been performed with 1,000–2,000 time steps for a single convective time unit (CTU), based on the MAC and freestream velocity, and the time-averaged quantities such as surface pressures and velocity components are collected over 5–15 CTUs after the transients in the forces and moments have been eliminated. The implicit system of equations is solved using a hierarchical adaptive nonlinear iteration method (HANIM), described in Ref. [61], which has resulted in 4–5 orders of magnitude in residual reduction in 6–8 subiterations for the problems presented here.

Figure 4 depicts profiles of integrated lift and drag coefficients as a function of angles of attack in the WMLES computations of the CRM-HL configuration. Compared to the experimental measurements, the WMLES solutions show reasonable agreement throughout the wide range of angles of attack. In particular, the computed lift coefficient at $\alpha = 19.57°$ that corresponds to the angle of attack near the experimental $C_{L,max}$ shows about 1% deviation from the measurement. For the post-stall cases, the WMLES lift predictions are consistent with the experimental data. In terms of the drag predictions shown in Fig. 4(b), WMLES solutions exhibit good matches with the experimental data, except for somewhat underpredicted drag in the highest angle-of-attack ($\alpha = 21.47°$) case. The discrepancy in drag in the 21.47° angle-of-attack case is possibly caused by the lack of tunnel-interference with the model in free-air simulations.



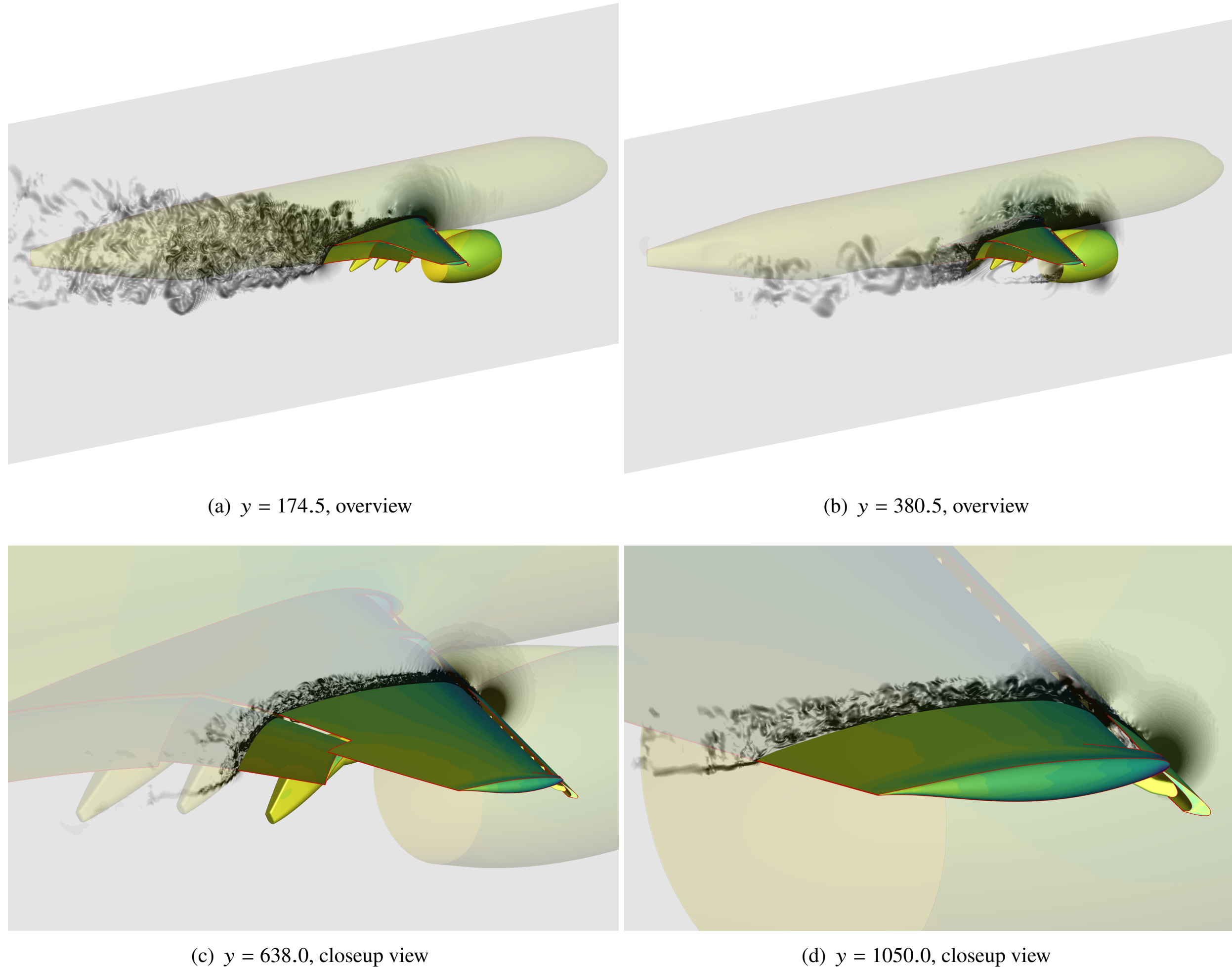
(a) Lift

(b) Drag

Fig. 4 Lift and drag polar profiles for NASA CRM-HL WMLES computations.

Figure 5 displays contours of instantaneous density-gradient magnitude obtained by the WMLES approach to visualize the basic flow structures around the CRM-HL configuration at the angle of attack of $\alpha = 19.57°$. Four spanwise stations are used, consisting of constant planes at $y = 174.5$, 380.5, 638.0 and 1050.0 inches, from inboard to outboard. At this critical angle of attack near the experimental maximum lift, due to complex interactions of the flows among the inboard slat, wing leading edge, fuselage-wing junctures, and nacelle, complicated vortical structures and turbulent eddies are generated and captured by the WMLES scheme. These shedding vortices propagate downstream and interact

with the main-wing and slat wakes, making the flow field highly unsteady. Progressing from the inboard to the outboard sections of the wing, the turbulent wakes behind the wing/flap become significantly smaller in the vertical direction. However, the trailing-edge slat wake is well captured above the main element and all the way to the suction side of the flap.



(a) y = 174.5, overview

(b) y = 380.5, overview

(c) y = 638.0, closeup view

(d) y = 1050.0, closeup view

Fig. 5 Contours of instantaneous density gradient magnitude of WMLES computations for NASA CRM-HL at four spanwise stations (19.57°)**; aircraft surface is colored by time-averaged pressures.**

Performance and the corresponding resources are provided in Table 5. The computational time in minutes is assessed for the solver to complete one CTU using 1000 steps per flow pass at the NASA Advanced Supercomputing (NAS) facility. The CPU runs use 200 AMD EPYC 7742 nodes (25,600 total cores). The GPU run for FLUDA uses 27 4 × NVIDIA V100 nodes with 36-core Intel Skylake 6154 CPUs (108 V100s). The FLUDA CPU implementation is observed to be 1.15× faster than the Fortran CPU version. The GPU setup enables a CTU to be simulated in about an hour. Performance is approximately commensurate with the hardware memory bandwidth ratio between the different architectures. One distinct difference for this example versus the other examples in this work is the use of the cell-based viscous discretization scheme rather than the edge-based viscous discretization scheme [48, 62]. In an initial evaluation, the edge-based viscous discretization scheme has shown an additional 20% acceleration (that is not included in the table) for the current problem; the scheme will be further investigated and assessed for WMLES applications.

Table 5 CRM-HL WMLES run times. The computational grid used in this assessment consists of 419 million points and 812 million mixed-type elements including hexahedra, tetrahedra, prisms, and pyramids. Wall time per convective time unit (CTU) is shown on various architectures. Speedup is device-normalized with respect to FLUDA AMD EPYC 7742. Vendor-reported maximum hardware memory bandwidth (MBW) is normalized with respect to AMD EPYC 7742.

Architecture	Implementation	Time [min] per CTU	Speedup	Hardware MBW Ratio
200 AMD EPYC 7742	Fortran	80	0.87	1.00
200 AMD EPYC 7742	FLUDA	69	1.00	1.00
27 (4 × NVIDIA V100)	FLUDA	60	2.13	2.20

C. Tilt Rotor Aeroacoustics Model

A rotor in hover conditions is used to demonstrate moving-grid capabilities across architectures. The Tilt Rotor Aeroacoustics Model (TRAM) is considered, which is a 25% V-22 three-bladed rotor with geometric and dynamic scaling. The model and geometry are detailed in Ref. [63]. The rotor radius, R, is 57 inches with a tip chord, c_{tip}, of 5.5 inches. The Reynolds number based on the tip chord is 2.1×10^6. The rotor collective for this example is 14°. The tip Mach number is 0.62.

A mixed-element grid similar to that employed for a previous study described in Ref. [64] is used here. A prismatic boundary layer mesh is used for the three blades, with a minimum wall spacing corresponding to approximately $y^+ \approx 1$. The wake grid has a characteristic spacing of 10% c_{tip}. The final grid is composed of 20,233,109 nodes and 97,302,140 elements, with approximately 10% of these elements comprising the prismatic boundary layer mesh immediately adjacent to the blade surfaces and tetrahedral elements away from the blade surfaces. A centerline slice of the grid is shown in Fig. 6. Although this mesh resolution is inadequate to fully resolve all relevant length scales, the current approach is sufficient to demonstrate relative computational performance across architectures.

The physical time step size corresponds to a 1° azimuthal rotation of the blades. Temporal integration is performed using a second-order backward-difference method. Ten subiterations are used, which provides a nominal reduction of four orders of magnitude in the unsteady residual at each time step. The edge-based viscous discretization described in Ref. [62] is used, and a Delayed Detached Eddy Simulation approach based on the one-equation model of Spalart and Allmaras is used for turbulence closure [46, 65, 66]. The mesh undergoes a rigid-body rotation at each time step through the use of a 4x4 matrix transform applied to the mesh coordinates as described in Ref. [67].

The hover figure of merit (FM) is computed using the thrust coefficient, C_T, and the torque coefficient, C_Q, as $\mathrm{FM} = \mathrm{C_T^{3/2}}/(\mathrm{C_Q}\sqrt{2})$. The value of FM converges after performing approximately four revolutions based on the initial flow field. Isosurfaces of the Q-criterion are shown in Fig. 7 and comparison of the computed FM value with experimental data taken from Ref. [68] is shown in Fig. 8. The figure of merit prediction on the current grid is within 2% of experimental data. Prior studies have shown improvements in accuracy with finer grid resolution and mesh adaptation approaches [64, 69].

Performance is compared across various architectures and is presented in Table 6. CPU runs for Fortran and FLUDA use 20 dual-socket AMD EPYC 7742 nodes with 2,560 total cores. GPU runs for FLUDA use 4 4 × NVIDIA V100 nodes with 36-core Intel Skylake 6154 CPUs (16 V100s). The time per revolution is about 19 minutes for the Fortran, 16 minutes for FLUDA on the same CPUs, and 9 minutes for the GPU-based approach. Normalizing to compare individual hardware, a V100 is approximately the same performance as 2.38 AMD EPYC 7742 nodes, which is slightly larger than the hardware memory bandwidth ratio. The GPU-based approach enables a converged value for the hover figure of merit to be obtained in approximately 34 minutes.

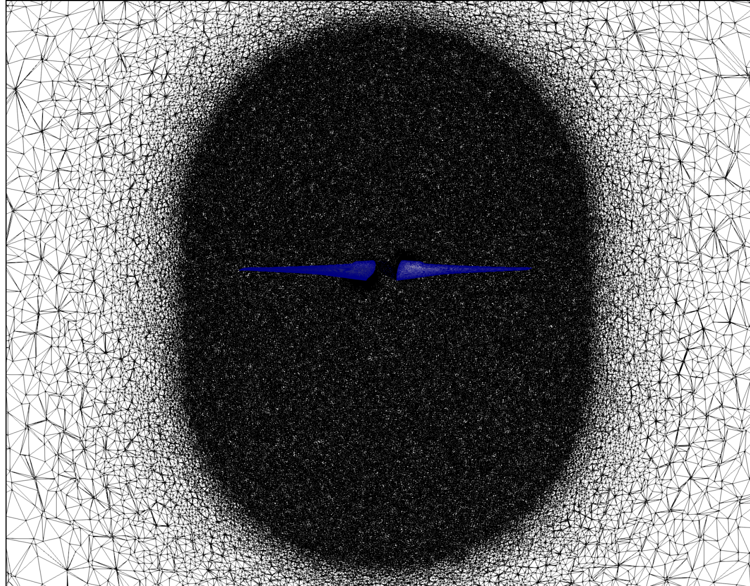


Fig. 6 Centerline slice of TRAM rotor volume grid with blades.

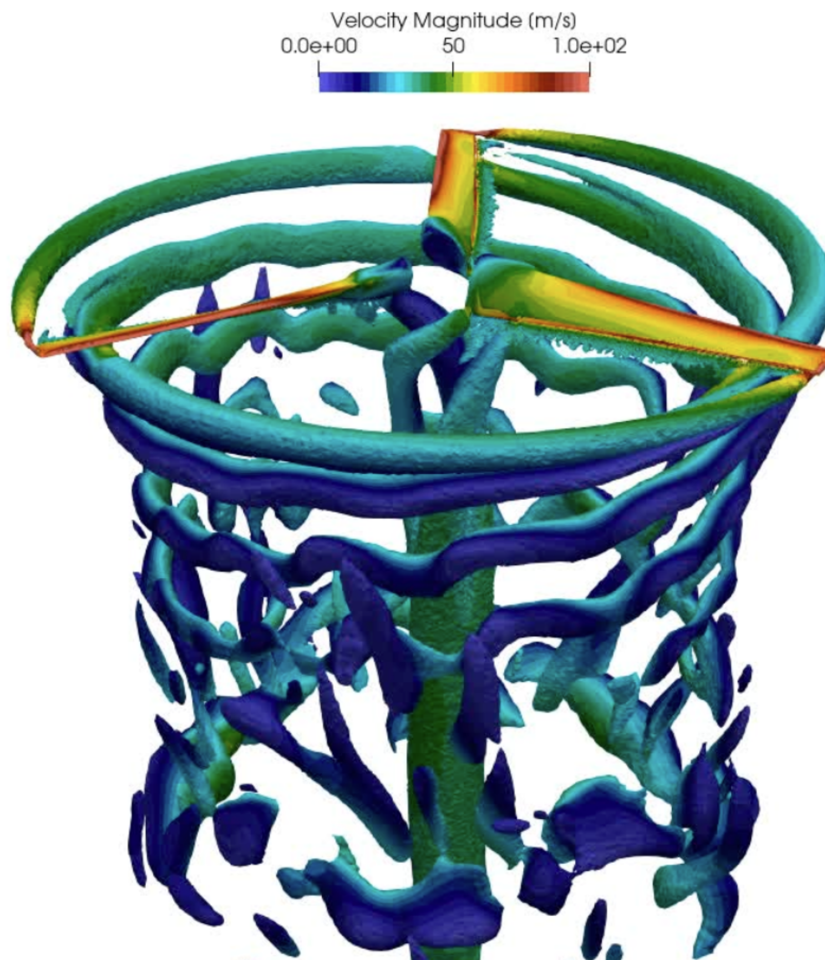


Fig. 7 Isosurfaces of Q-criterion (0.001 1/s) colored by velocity magnitude [m/s] for the TRAM rotor.

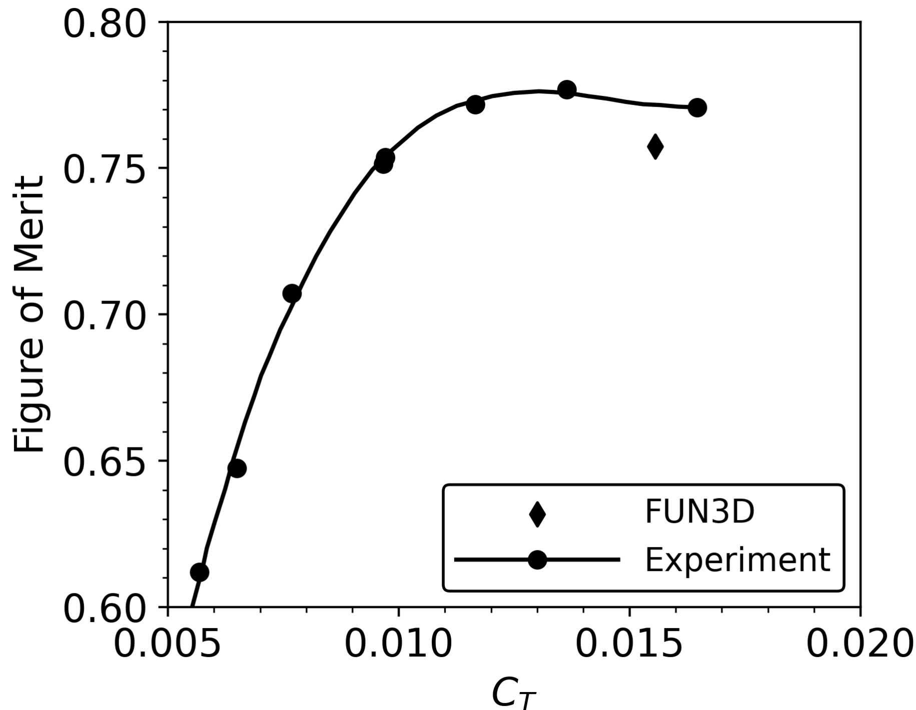


Fig. 8 Figure of merit versus thrust coefficient with experimental data taken from Ref. [68].

Table 6 Time per revolution for the TRAM rotor on various architectures. Speedup is device-normalized with respect to FLUDA AMD EPYC 7742. Vendor-reported maximum hardware memory bandwidth (MBW) is normalized with respect to AMD EPYC 7742.

Architecture	Implementation	Time per Revolution [min]	Speedup	Hardware MBW Ratio
20 AMD EPYC 7742	Fortran	19.2	0.84	1.00
20 AMD EPYC 7742	FLUDA	16.2	1.00	1.00
4 (4 × NVIDIA V100)	FLUDA	8.5	2.38	2.20

D. High-Speed Flow: Crew Exploration Vehicle

The flow over the Crew Exploration Vehicle (CEV) is considered. Specifically, a high-enthalpy test performed in the CUBRC LENS-I facility in air [70] is investigated using a model with a radius, R, of 2.87 inches. In addition to experimental pressure and heat transfer on the surface, NASA Data-Parallel Line Relaxation (DPLR) [71] data are also available from Ref. [72] for further comparisons. Run 10, the highest enthalpy condition (H_0 = 12.4 MJ/kg), is simulated. A 5-species, two-temperature gas model is used as in the DPLR simulation. The surface is modeled as a noncatalytic cold ($T = T_V$ = 300 K) no-slip wall. The freestream temperature, assumed identical for both translational/rotational and vibrational temperatures, density, and velocity are 631 K, $1.08{\times}10^{-3}$ kg/m^3, and 4601 m/s, respectively. The freestream mass fractions are $\vec{Y} = Y_{N_2}, Y_{O_2}, Y_{NO}, Y_O, Y_N = 0.7377, 0.1387, 0.0590, 0.0646, 0.0000$. The angle of attack is 28°. Laminar flow is assumed. Edge-based diffusion methods [62] are used, which is shown to accelerate time to solution by over a factor of two for a similar CPU run in the reference. Steady-state time stepping is employed.

A hybrid grid adaptation strategy is utilized following the approach in Ref. [73], where a fixed surface and prismatic boundary layer are employed with NASA *refine* [74] adapting the remaining tetrahedra to the translational/rotational temperature Hessian. An initial unstructured grid is generated using Capstone [75]. The heatshield surface spacing is 2% R, with the shoulder refined at 1% R. The boundary layer is extruded with an initial wall spacing of 2×10^{-6} R with

40 layers and a growth rate of 1.2. The total boundary layer thickness is ≈1.5% R with aspect ratios of 10 at the edge of the boundary layer. The aspect ratios near the wall are 5,000 to 10,000. The initial grid consists of 1,837,670 points, 1,130,796 tetrahedra, and 3,250,960 prisms. The target number of points is 3 million. Fifteen *refine* adaptation cycles are performed. The final grid after 15 cycles consists of 3,049,876 points, 8,270,063 tetrahedra, and 3,250,960 prisms. The initial and final grids are shown in Fig. 9. In total, 21,000 iterations are performed, with initial cycles running more iterations to initialize the flow field. Later cycles run only 1,000 iterations each. Final iterative convergence is 4-5 orders of magnitude for all equations without limiter freezing. Heat transfer magnitudes were observed unchanged by subsequent iterations and cycles.

Figure 10 depicts temperature contours for the capsule. The translational/rotational temperature peaks at 9700 K at the shock and is smoothly captured due to grid adaptation. At this enthalpy, molecular oxygen dissociates completely post shock and in the wake. Molecular nitrogen is partially dissociated, leading to significant nitric oxide at the shock front. The bow shock and wake are captured well with anisotropic tetrahedra even using only 3 million total points in the entire domain. Figure 11 plots surface quantities of interest. All quantities are smoothly captured on the unstructured grid. The cell Reynolds number based on speed of sound is below one on the entire surface. Comparisons to experiment and DPLR are shown in Fig. 12. Surface properties are smooth along the heatshield and backshell of the capsule. Pressures agree extremely well with DPLR, but overpredict experiment slightly at the center and rear shoulder of the heatshield. Heat transfer is slightly overpredicted versus experiment for both FUN3D and DPLR. FUN3D predicts slightly larger heat transfer than DPLR. The shoulder heat transfer is more smoothly captured for FUN3D; it is unknown if grid adaptation was used for the DPLR results. Further grid refinement is likely necessary for a more rigorous comparison.

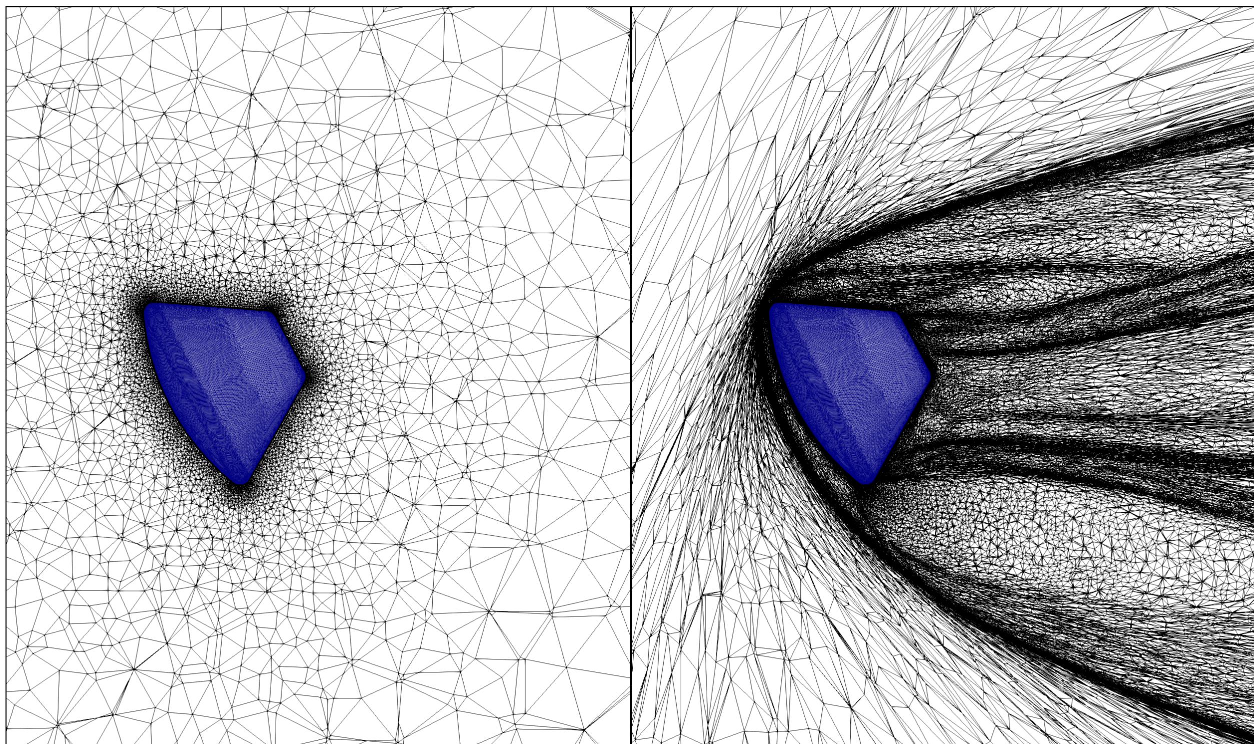


Fig. 9 Left: Initial CEV grid. Right: Final adapted CEV grid. The surface and boundary layer grids are fixed.

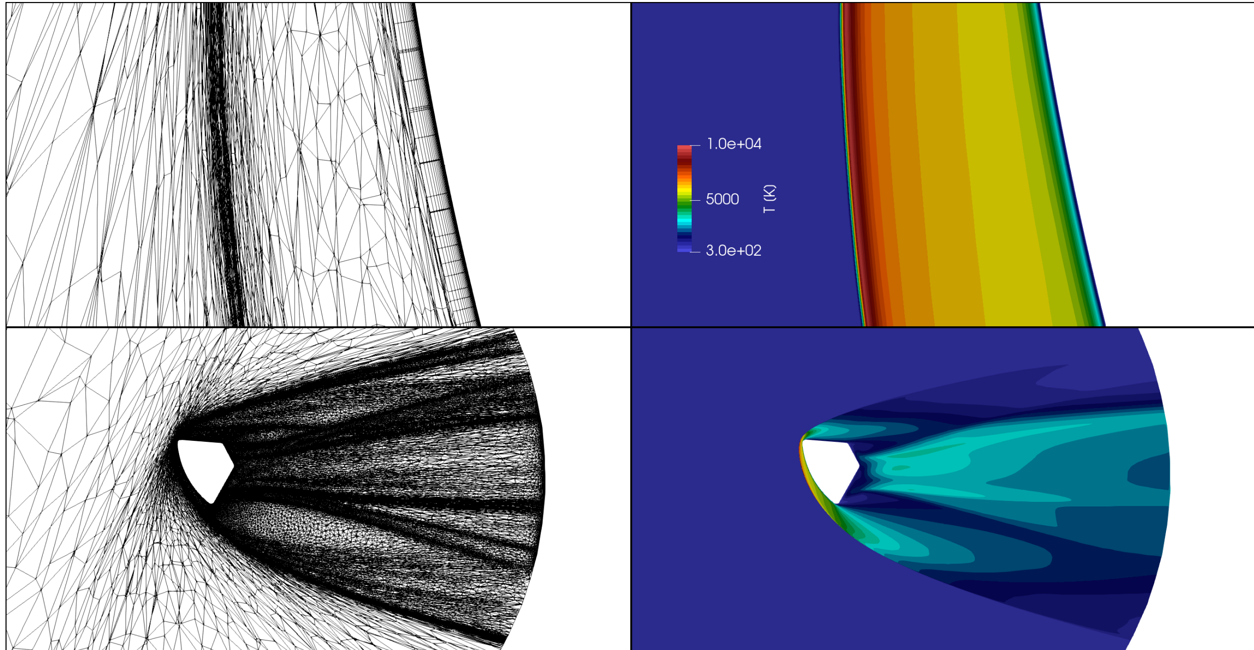


Fig. 10 Left: Final CEV grid near the heatshield and on the centerline. Right: Translational/rotational temperature contours in Kelvin.

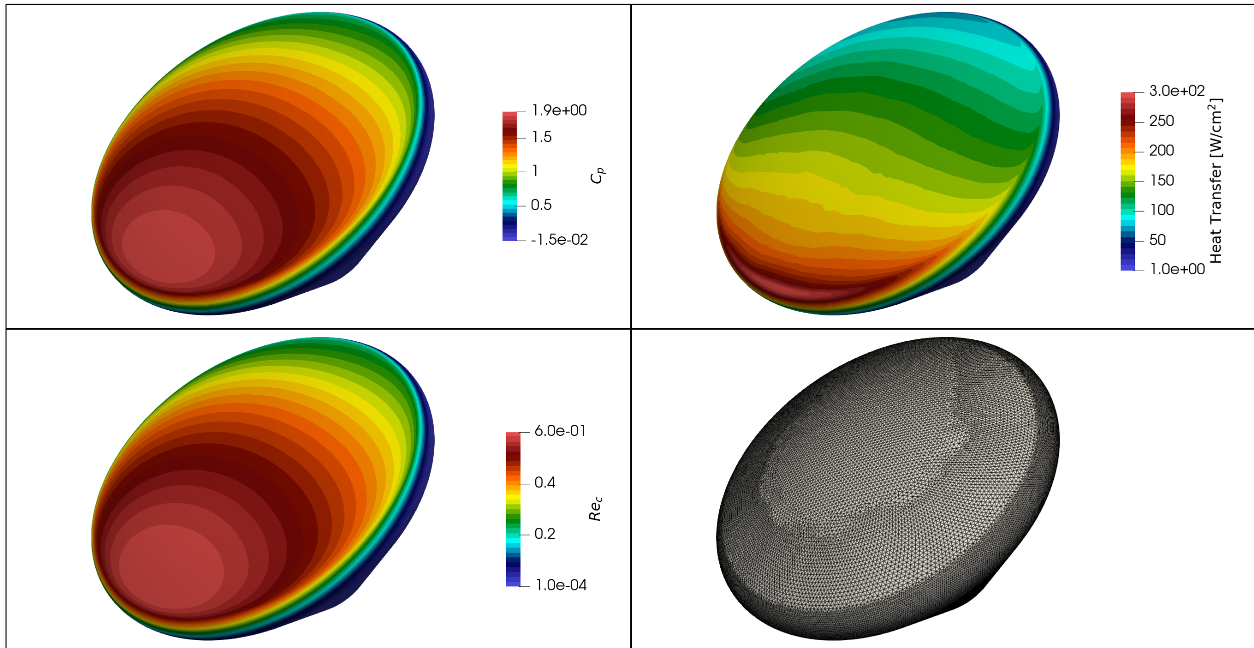


Fig. 11 CEV surface contours for pressure coefficient, heat transfer, cell Reynolds number, and grid.

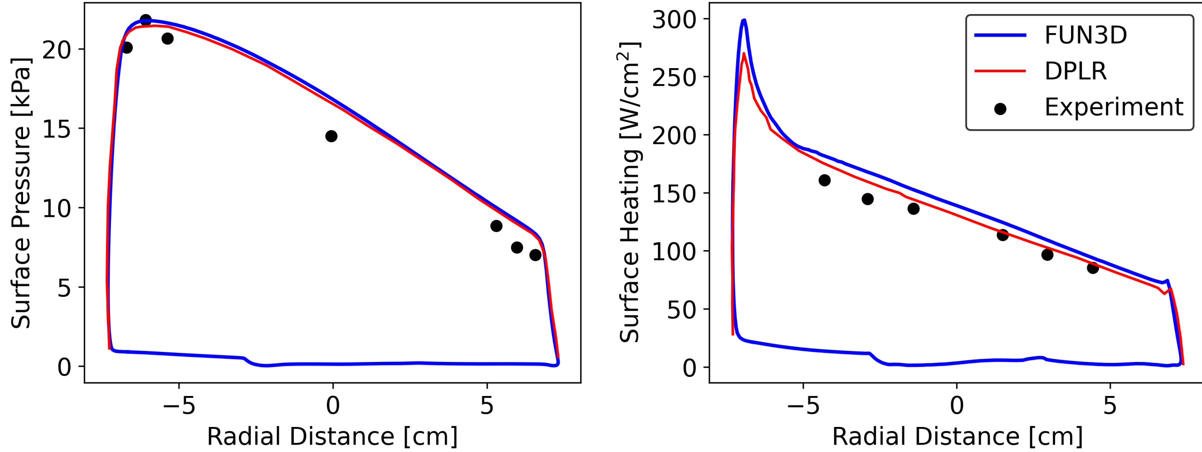


Fig. 12 CEV comparisons to DPLR and experiment. Left: Surface pressure along the centerline. Right: Heat transfer along the centerline.

Performance is compared across various architectures. CPU runs for Fortran and FLUDA use 15 dual-socket AMD EPYC 7742 nodes with 1,920 total cores. GPU runs for FLUDA use 4 × NVIDIA V100 nodes with 36-core Intel Skylake 6154 CPUs and a 4 × NVIDIA 40 GB A100 node with 128-core AMD EPYC 7742 CPUs. The case is also run on a single NVIDIA 80 GB A100 GPU. The A100 GPU simulations are run using 8 ranks per GPU using the CUDA Multi-process Service (MPS) to improve (CPU) I/O performance. Simulation performance is presented in Table 7. NASA *refine* is run on only the CPUs using 10 adaptation passes per execution. Preprocessing and postprocessing is also tabulated as refinement is also file based at this time. GPU-enabled in-core refinement should improve performance for this approach. The Fortran implementation on 15 EPYC nodes completes the CFD in 104 minutes. FLUDA shows ≈30% speedup over Fortran, which is higher than that observed for perfect gas flows. We hypothesize much of this improvement is due to compile-time values provided by C++ templates that are unavailable in Fortran. The 8 V100 GPUs complete the CFD in 59 minutes. The 4 A100 GPUs complete the CFD in 79 minutes. A single A100 GPU completes the CFD in 262 minutes. Performance is generally commensurate with hardware memory bandwidth ratio. Overall, a 3D capsule simulation workflow including wake using a 5-species two-temperature model and *refine* can be run on a machine with only 1 NVIDIA A100 GPU in ≈6 hours.

One important detail to note is that these simulations are performed using edge-based viscous terms [48]. For cell-based viscous terms, which have been the primary viscous terms in FUN3D for the last two decades and are still the default as of this writing, FLUDA sees significantly higher speedups over Fortran due to templating and the reduction of state variables. The edge-based viscous terms perform much better on both CPUs and GPUs, leaving a larger percentage of the execution time to the linear solver, which is strongly memory-bound.

Table 7 CEV run times. I/O includes FUN3D preprocessing and postprocessing required for visualization and *refine*. The NVIDIA A100 GPU runs use CUDA MPS with 8 MPI ranks. Multiple ranks per GPU generally incurs a 5-10% compute overhead, but speeds up other non-GPU-enabled code such as I/O. The V100 GPU results use a single MPI rank per GPU. The NVIDIA V100 nodes have 36-core Skylake 6154 CPUs. The NVIDIA A100 nodes have 128-core dual-socket AMD EPYC 7742 CPUs. CFD speedup is device-normalized with respect to FLUDA AMD EPYC 7742. Vendor-reported maximum hardware memory bandwidth (MBW) is normalized with respect to AMD EPYC 7742.

Architecture	Implementation	Total [min]	*refine* [min]	I/O [min]	CFD [min]	Speedup	Hardware MBW Ratio
15 AMD EPYC 7742	Fortran	126	11	11	104	0.76	1.00
15 AMD EPYC 7742	FLUDA	101	11	11	79	1.00	1.00
2 (4 × NVIDIA V100)	FLUDA	162	82	21	59	2.51	2.20
4 × NVIDIA 40 GB SXM A100	FLUDA	150	57	14	79	3.75	3.80
1 × NVIDIA 80 GB A100	FLUDA	337	57	18	262	4.52	4.72

E. High-Speed Flow: Dream Chaser

A second hypersonic benchmark is carried out on a more complex vehicle shape: the Sierra Space Dream Chaser© spaceplane. The freestream reentry conditions are 5 km/s or Mach 15.8 and an angle of attack of 40° with zero side slip. The wall boundaries of Dream Chaser are modeled as reaction cured glass (RCG) coating and assumed in radiative equilibrium. Laminar flow is modeled using a 5-species, one-temperature model. Results are compared to those of DPLR, which is considered a gold standard hypersonic structured CFD code.

The initial unstructured grid used for FUN3D consists of 8,027,447 points and 25,734,402 cells. The surface spacing is roughly 0.0254 meters and the first cell height is specified as 5.0×10^{-6} meters. The anisotropic viscous grid is generated with 35 layers and 10,911,866 prismatic cells. The remaining volume grid consists of 14,822,130 tetrahedra cells and just 406 pyramids. The number of adapted points is set to 4 million and the final grid consists of 12,238,724 points and 39,418,400 tetrahedral cells (with the identical number of prisms and pyramids as the original grid). The DPLR multiblock structured grid consists of 29,441,082 points and 28,586,880 hexahedra cells. The DPLR grid is adapted four times to gain full convergence of heat flux. Figures 14–16 show both solver's grids before and after adaptation, along with a close-up of the grid at the nose of the vehicle.

The results from FUN3D after adaptation are in good agreement with DPLR. The maximum stagnation pressure for FUN3D is 0.18% higher than DPLR. Figure 17 shows the pressure contours and centerline comparisons. The centerline pressures agree well and differences in the contours are nearly indistinguishable. Figure 18 compares the heat flux results. The FUN3D maximum heating is 1.9% higher than DPLR. There is minor variation in heat flux near the stagnation point for FUN3D's result, which may suggest that the solution is not fully converged or that several more adaptation cycles are required. Note that FUN3D with *refine*'s shock adaptation captures secondary shocks as seen in FUN3D's final grid in Fig. 14. Furthermore, refinement occurs across the flow field and is not limited strictly to shock refinement, which other refinement methods commonly employ [76, 77]. Since DPLR only adapts along k-lines to capture the bow shock, it cannot pick up secondary shocks.

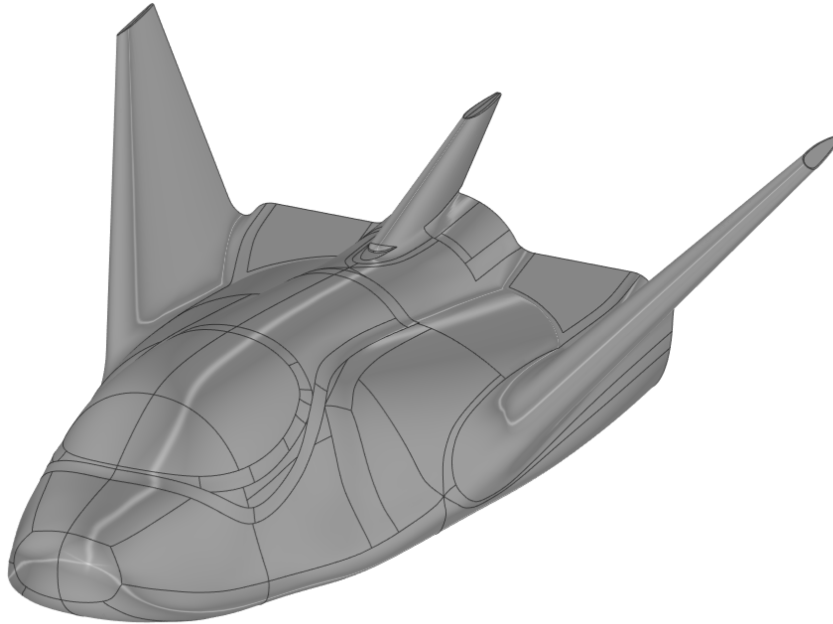


Fig. 13 Sierra Space's Dream Chaser outer mold line.
[***Credit: Sierra Space Corporation***]

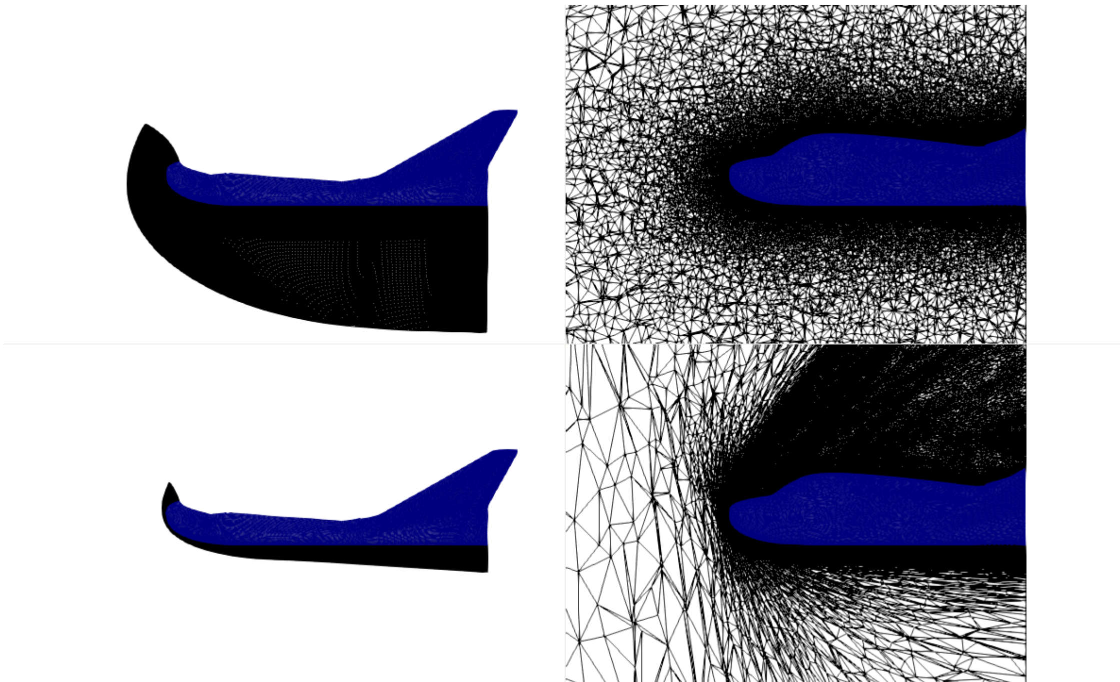


Fig. 14 Comparison of the initial and final DPLR and FUN3D grids.
[*Credit: Sierra Space Corporation*]

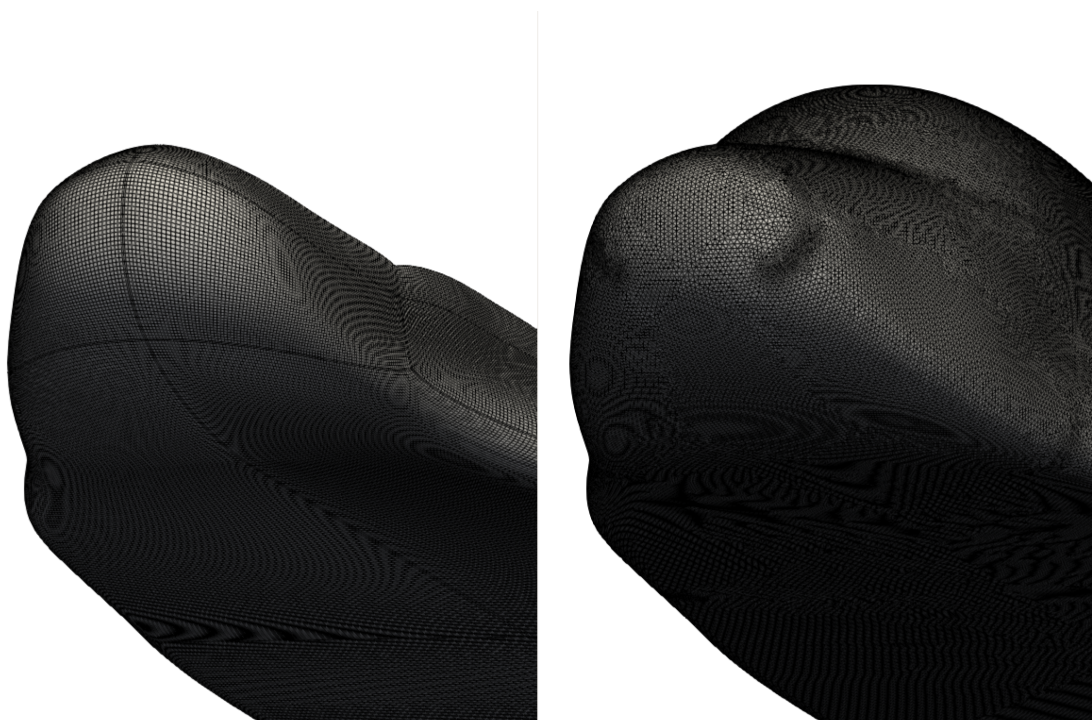


Fig. 15 Comparison of the surface grids for DPLR and FUN3D.
[*Credit: Sierra Space Corporation*]

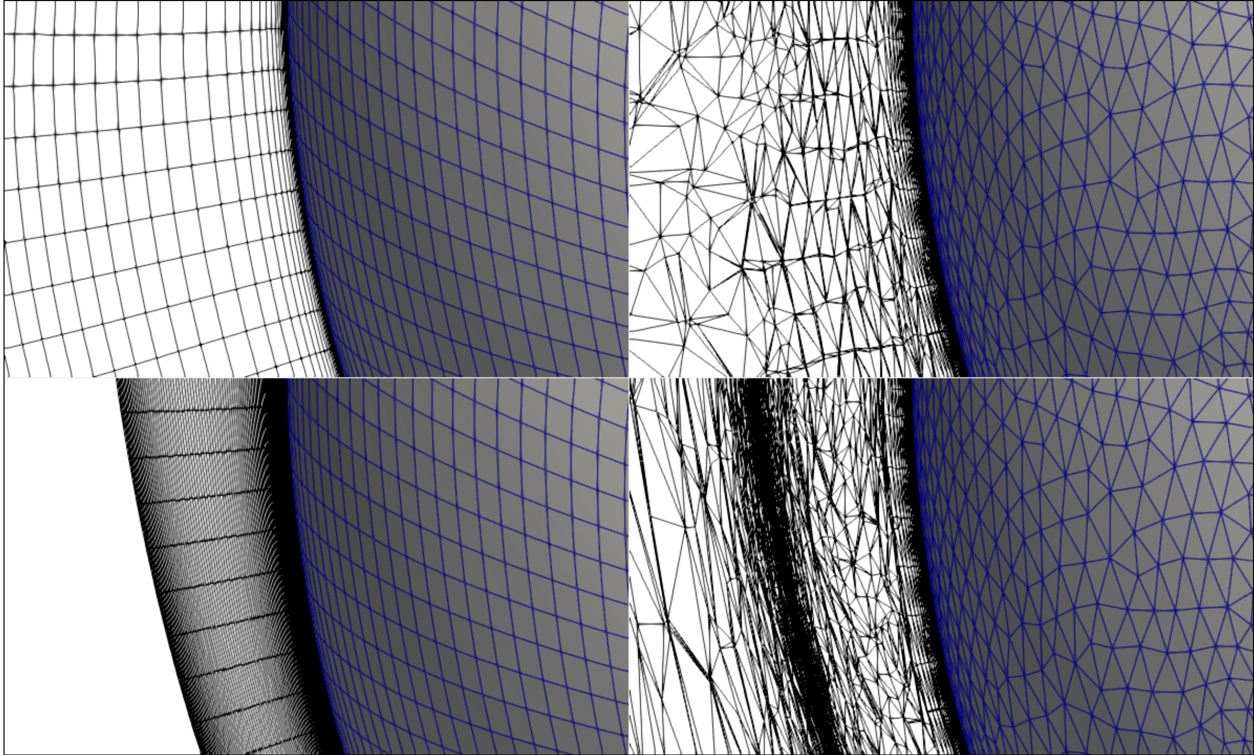


Fig. 16 Close-up comparison of the DPLR and FUN3D grids on the nose.
[*Credit: Sierra Space Corporation*]

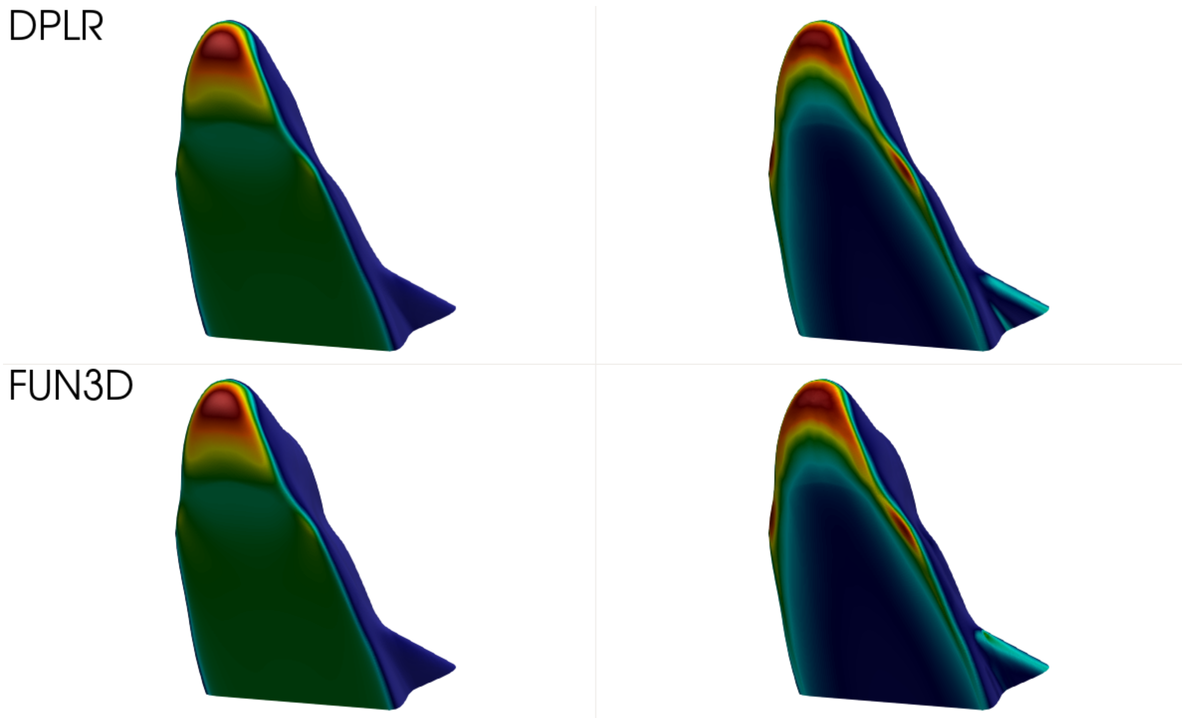


Fig. 17 Pressure and heating contours for DPLR and FUN3D.
[*Credit: Sierra Space Corporation*]

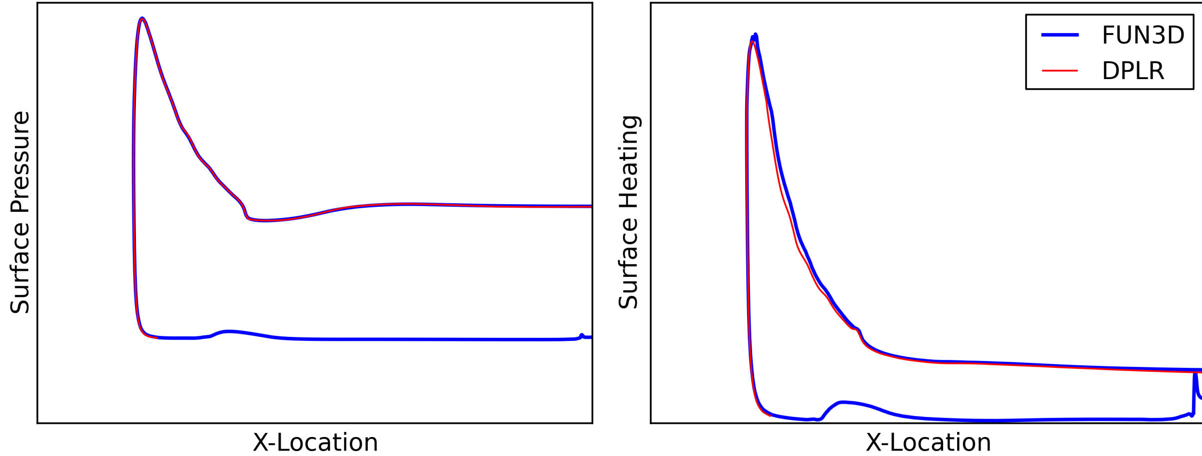


Fig. 18 Pressure and heating along the centerline for DPLR and FUN3D.
[***Credit: Sierra Space Corporation***]

CPU simulations are performed on 30 AMD EPYC 7742 nodes (3,840 cores). FUN3D GPU simulations are performed on 4 4 × NVIDIA V100 nodes with 36-core Intel Skylake 6154 CPUs (16 V100s). Fifteen *refine* cycles are performed for the FUN3D approach. In total, 21,000 iterations are run across the cycles, with the initial cycles running more iterations to ensure smooth shock fronts for successful adaptations. Table 8 provides run time information for the different architectures and implementations. Similar to the CEV example, I/O and *refine* are separated to emphasize the CFD performance. The wall time is several hours for both architectures. FLUDA on the CPU is 1.20× faster than the legacy Fortran implementation, attributed primarily due to templating on the gas model physics (e.g., the number of species). The CFD on the four GPU nodes is 1.14× faster than 30 CPU nodes. CFD performance is commensurate with hardware memory bandwidth ratio as observed in other sections. While this case may be run on fewer GPUs (1 node), the I/O and *refine* overhead become very large and necessitate alternate strategies such as running *refine* separately using additional CPU nodes. Comparisons to DPLR are shown in Table 9; run times are comparable. We note that the discretization and solution algorithms for the two solvers differ.

Table 8 Dream Chaser run times. I/O includes FUN3D preprocessing and postprocessing required for visualization and *refine*. A single MPI rank per GPU is utilized. The NVIDIA V100 nodes have 36-core Skylake CPUs. CFD speedup is device-normalized with respect to FLUDA AMD EPYC 7742. Vendor-reported maximum hardware memory bandwidth (MBW) is normalized with respect to AMD EPYC 7742.
[***Credit: Sierra Space Corporation***]

Architecture	Implementation	Total [min]	*refine* [min]	I/O [min]	CFD [min]	Speedup	Hardware MBW Ratio
30 AMD EPYC 7742	Fortran	132	24	7	101	0.83	1.00
30 AMD EPYC 7742	FLUDA	115	24	7	84	1.00	1.00
4 (4 × NVIDIA V100)	FLUDA	293	180	39	74	2.13	2.20

Table 9 Dream Chaser simulation performance for FUN3D and DPLR on 30 AMD EPYC 7742 nodes (3,840 cores). CFD time includes preprocessing and postprocessing time (I/O).
[***Credit: Sierra Space Corporation***]

Code	DOFs	Iterations	refinement [min]	CFD [min]	Total [min]
DPLR	28.6M	7000	6	52	58
FLUDA CPU	12.2M	21000	24	91	115

V. Summary

A multi-architecture approach for implicit CFD on unstructured grids has been described and results have been demonstrated across the speed range from low subsonic conditions for an aircraft in a high-lift configuration to hypersonic reentry vehicles. A thin abstraction layer above CUDA C++ similar to AMD's HIP is employed. Diverging architecture-specific code is minimal and comprises less than 2% of the total lines of code. The abstractions are roughly 500 lines of code and enable NVIDIA GPU usage with CUDA C++ with only an NVIDIA software stack, CPU usage with only an ISO C++ compiler, AMD GPU usage with AMD HIP, and Intel GPU usage with SYCL. CPU results using the multi-architecture GPU-first approach are generally faster than the legacy Fortran implementation. Results on various architectures show that performance scales with memory bandwidth as expected of a code with low arithmetic intensity. Performant GPU-enabled CFD software enables design cycles and database generation to occur using less time and fewer resources for both engineers using a single workstation and those with access to large computing clusters.

Acknowledgments

Assistance from the technical staff at Advanced Micro Devices, Inc., Intel Corporation, and NVIDIA Corporation is greatly appreciated. Support from the HPE/Cray Frontier Center of Excellence, Oak Ridge Leadership Computing Facility (OLCF) Frontier Center for Accelerated Application Readiness (CAAR) program, and the Argonne Leadership Computing Facility (ALCF) are also acknowledged. This research used resources of the OLCF 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. This research also used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract No. DE-AC02-06CH11357. The authors would like to thank Christopher Stone for his contributions to AMD HIP extensions, optimizations, and the autotuning framework. The authors would like to thank Sameer Shende of the University of Oregon. The authors would also like to recognize the support of Intel Corporation through the Intel oneAPI Center of Excellence located at Old Dominion University. This research was sponsored by the NASA Langley Research Center CIF/IRAD program, and the NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate.

References

[1] NVIDIA Corporation, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2020. Last Accessed May 2, 2022.

[2] Kincade, Kathy, "Berkeley Lab, Oak Ridge, NVIDIA Team Breaks Exaop Barrier With Deep Learning Application," https://www.nersc.gov/news-publications/nersc-news/science-news/2018/berkeley-lab-oak-ridge-nvidia-team-breaks-exaop-barrier-with-deep-learning-application/, 2018. Last Accessed May 2, 2022.

[3] ORNL, "Summit Simulates how Humans will 'Brake' During Mars Landing," https://www.ornl.gov/news/summit-simulates-how-humans-will-brake-during-mars-landing, 2019. Last Accessed May 2, 2022.

[4] Strohmaier, E., Dongarra, J., Simon, H., Meuer, M., and Meuer, H., "The Top 500 List," http://www.top500.org, 2022. Last Accessed November 14, 2022.

[5] Stevens, R., Ramprakash, J., Messina, P., Papka, M., and Riley, K., "Aurora: Argonne's Next-Generation Exascale Supercomputer," Tech. rep., Argonne National Laboratory, 2019.

[6] ORNL, "Frontier," https://www.olcf.ornl.gov/frontier, 2020. Last Accessed May 2, 2022.

[7] Thomas, J., "El Capitan," https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer, 2020. Last Accessed May 2, 2022.

[8] Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D., "CFD Vision 2030 Study: A Path to Revolutionary Computational Aerosciences," *NASA CR-2014-218178*, 2014.

[9] Stone, J. E., Gohara, D., and Shi, G., "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, Vol. 12, No. 3, 2010, p. 66.

[10] AMD, "AMD ROCm Platform," https://rocmdocs.amd.com/en/latest/, 2022. Last Accessed May 2, 2022.

[11] "CHIP-SPV," https://github.com/CHIP-SPV/chip-spv, 2022. Last Accessed May 2, 2022.

[12] Intel, "oneAPI Level Zero," https://spec.oneapi.io/level-zero/latest/index.html, 2022. Last Accessed May 10, 2022.

[13] The OpenMP Consortium, "The OpenMP API," https://www.openmp.org, 2020. Last Accessed May 2, 2022.

[14] The OpenACC Organization, "OpenACC," https://www.openacc.org, 2022. Last Accessed May 2, 2022.

[15] Keryell, R., Reyes, R., and Howes, L., "Khronos SYCL for OpenCL: a tutorial," *Proceedings of the 3rd International Workshop on OpenCL*, 2015.

[16] Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., and Tian, X., *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*, Springer Nature, 2021.

[17] Codeplay, "ComputeCpp," https://www.codeplay.com/products/computecpp, 2022. Last Accessed May 2, 2022.

[18] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3202–3216.

[19] Hornung, R. D., and Keasler, J. A., *The RAJA Portability Layer: Overview and Status*, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2014.

[20] NVIDIA, "Developing Accelerated Code with Standard Language Parallelism," https://developer.nvidia.com/blog/developing-accelerated-code-with-standard-language-parallelism, 2022. Last Accessed May 2, 2022.

[21] Sathre, P., Gardner, M., and Feng, W.-c., "On the Portability of CPU-accelerated Applications via Automated Source-to-source Translation," *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2019, pp. 1–8.

[22] Thavappiragasam, M., Elwasif, W., and Sedova, A., "Portability for GPU-accelerated Molecular Docking Applications for Cloud and HPC: Can Portable Compiler Directives Provide Performance across all Platforms?" *arXiv preprint arXiv:2203.02096*, 2022.

[23] Evans, T. M., Siegel, A., Draeger, E. W., Deslippe, J., Francois, M. M., Germann, T. C., Hart, W. E., and Martin, D. F., "A Survey of Software Implementations used by Application Codes in the Exascale Computing Project," *The International Journal of High Performance Computing Applications*, 2021.

[24] Biedron, R. T., Carlson, J.-R., Derlaga, J. M., Gnoffo, P. A., Hammond, D. P., Jones, W. T., Kleb, B., Lee-Rausch, E. M., Nielsen, E. J., Park, M. A., et al., *FUN3D Manual: 13.7*, NASA TM 20205010139, 2020.

[25] Walden, A., Nielsen, E. J., Zubair, M., Linford, J. C., Wohlbier, J. G., Luitjens, J. P., Orender, J., Beekman, I., Khuvis, S., and Shende, S. S., "Unstructured-Grid CFD Algorithms on Many-Core Architectures," *Technical Research Posters of International Supercomputing Conference for High Performance Computing, Networking, Storage, and Analysis. SC*, 2017, p. 17.

[26] Walden, A., Nielsen, E., Korzun, A., et al., "Landing on Mars: Petascale Unstructured Grid Computational Fluid Dynamics on Summit," *International Workshop on OpenPOWER for HPC*, ISC, 2019, p. 87.

[27] Nastac, G., Walden, A., Nielsen, E. J., and Frendi, A., "Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs," *AIAA Paper 2021-0159*, 2021.

[28] ORNL, "Summit," https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit, 2020. Last Accessed May 2, 2022.

[29] Korzun, A., Nastac, G., Walden, A., Nielsen, E. J., Jones, W., and Moran, P., "Application of a Detached Eddy Simulation Approach with Finite-Rate Chemistry to Mars-Relevant Retropropulsion Operating Environments," *AIAA Paper 2022-2298*, 2022.

[30] Nastac, G., Korzun, A., Walden, A., Nielsen, E. J., Jones, W., and Moran, P., "Computational Investigation of the Effect of Chemistry on Mars Supersonic Retropropulsion Environments," *AIAA Paper 2022-2299*, 2022.

[31] Zubair, M., Nielsen, E., Luitjens, J., and Hammond, D., "An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units," *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA4)*, IEEE, 2016, pp. 18–25.

[32] Walden, A. C., Zubair, M., and Nielsen, E. J., "Performance and Portability of a Linear Solver Across Emerging Architectures," *International Workshop on Accelerator Programming Using Directives*, Springer, 2020, pp. 61–79.

[33] Walden, A., Zubair, M., Stone, C. P., and Nielsen, E. J., "Memory Optimizations for Sparse Linear Algebra on GPU Hardware," *2021 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, IEEE, 2021, pp. 25–32.

[34] McBride, B. J., Gordon, S., and Reno, M. A., "Coefficients for Calculating Thermodynamic and Transport Properties of Individual Species," *NASA TM 4513*, 1993.

[35] Gnoffo, P. A., Gupta, R. N., and Shinn, J. L., *Conservation Equations and Physical Models for Hypersonic Air Flows in Thermal and Chemical Nonequilibrium*, NASA TP 2867, 1989.

[36] Park, C., "Assessment of Two-Temperature Kinetic Model for Ionizing Air," *Journal of Thermophysics and Heat Transfer*, Vol. 3, No. 3, 1989, pp. 233–244.

[37] Burg, C., "Higher Order Variable Extrapolation For Unstructured Finite Volume RANS Flow Solvers," *AIAA Paper 2005-4999*, 2005.

[38] Liu, Y., Diskin, B., Anderson, W. K., Nielsen, E., and Wang, L., "Edge Based Viscous Method for Node-Centered Formulations," *AIAA Paper 2021-2728*, 2021.

[39] The MPI Forum, "The MPI Forum Website," `http://www.mpi-forum.org`, 2020. Last Accessed May 2, 2022.

[40] Zhang, X., Jones, W. T., Wood, S. L., and Park, M. A., "Component-Based Development of CFD Software FUN3D," *AIAA Paper 2022-0253*, 2022.

[41] Jones, W. T., Wood, S. L., Jacobson, K., and Anderson, W. K., "Interoperable Application Programming Interfaces for Computer Aided Engineering Applications," *AIAA Paper 2021-1364*, 2021.

[42] Anderson, W. K., Newman, J. C., and Karman, S. L., "Stabilized Finite Elements in FUN3D," *Journal of Aircraft*, Vol. 55, No. 2, 2018, pp. 696–714.

[43] NVIDIA, "CUDA C++ Programming Guide: Performance Guidelines," `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#performance-guidelines`, 2022. Last Accessed May 10, 2022.

[44] Stone, C. P., Walden, A., Zubair, M., and Nielsen, E. J., "Accelerating Unstructured-grid CFD Algorithms on NVIDIA and AMD GPUs," *2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, IEEE Computer Society, 2021, pp. 19–26.

[45] Vassberg, J., Tinoco, E., Mani, M., Rider, B., Zickuhr, T., Levy, D., Brodersen, O., Eisfeld, B., Crippa, S., Wahls, R., et al., "Summary of the fourth AIAA CFD drag prediction workshop," *AIAA Paper 2010-4547*, 2010.

[46] Spalart, P., and Allmaras, S., "A One-Equation Turbulence Model for Aerodynamic Flows," *30th Aerospace Sciences Meeting and Exhibit 1992-439*, 1992.

[47] Spalart, P. R., "Strategies for Turbulence Modelling and Simulations," *International Journal of Heat and Fluid Flow*, Vol. 21, No. 3, 2000, pp. 252–263.

[48] Liu, Y., Diskin, B., Nishikawa, H., Anderson, W. K., Nastac, G., Nielsen, E., Walden, A., and Wang, L., "Assessment of Edge-Based Viscous Method for Corner-Flow Solutions on Graphics Processing Units," *AIAA Paper 2023*, 2023.

[49] NASA, "FUN3D Computational Performance," `https://fun3d.larc.nasa.gov/example-21.html`, 2009. Last Accessed November 2, 2022.

[50] "NASA HECC: Skylake Processors," `https://www.nas.nasa.gov/hecc/support/kb/skylake-processors_550.html`, 2021. Last Accessed November 2, 2022.

[51] AMD, "AMD EPYC 7742," `https://www.amd.com/en/products/cpu/amd-epyc-7742`, 2021. Last Accessed November 2, 2022.

[52] NVIDIA, "NVIDIA V100," `https://www.nvidia.com/en-us/data-center/v100/`, 2022. Last Accessed November 2, 2022.

[53] NVIDIA, "NVIDIA A100 40GB PCIe GPU Accelerator," `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/A100-PCIE-Prduct-Brief.pdf`, 2020. Last Accessed November 2, 2022.

[54] AMD, "AMD Instinct MI210 Accelerator," https://www.amd.com/en/products/server-accelerators/amd-instinct-mi210, 2020. Last Accessed November 2, 2022.

[55] Kawai, S., and Larsson, J., "Wall-Modeling in Large Eddy Simulation: Length Scales, Grid Resolution, and Accuracy," *Physics of Fluids*, Vol. 24, No. 1, 2012, p. 015105.

[56] Spalding, D. B., "A Single Formula for the "Law of the Wall"," *J. Appl. Mech.*, Vol. 28, No. 3, 1961, pp. 455–458.

[57] Wang, L., Anderson, W. K., Nielsen, E. J., Balakumar, P., Park, M. A., Carlson, J.-R., Iyer, P. S., and Diskin, B., "Wall-Modeled Large-Eddy Simulations for High-Lift Configurations using FUN3D (Invited)," AIAA Paper 2022-1555, 2022. https://doi.org/10.2514/6.2022-1555.

[58] Wang, L., Anderson, W. K., Nielsen, E. J., Iyer, P. S., and Diskin, B., "Wall-Modeled Large Eddy Simulation Method for Unstructured-Grid Navier-Stokes Solvers (invited)," AIAA SciTech 2023 Forum, 2023.

[59] Evans, A. N., Lacy, D. S., Smith, I., and Rivers, M. B., "Test Summary of the NASA High-Lift Common Research Model Half-Span at QinetiQ 5-Metre Pressurized Low-Speed Wind Tunnel," AIAA Paper 2020–2770, 2020.

[60] NASA and AIAA, "High-Lift Prediction Workshop," , May 2, 2022 [Online]. URL https://hiliftpw.larc.nasa.gov/Workshop4/testcases.html.

[61] Wang, L., Diskin, B., Nielsen, E., and Liu, Y., "Improvements in Iterative Convergence of FUN3D Solutions," *AIAA Paper 2021-0857*, 2021.

[62] Liu, Y., Diskin, B., Nishikawa, H., Anderson, W. K., Nastac, G., Nielsen, E. J., and Wang, L., "Edge-Based Viscous Method for Mixed-Element Node-Centered Finite-Volume Formulation," *AIAA Paper 2022-4083*, 2022.

[63] Johnson, W., "Calculation of Tilt Rotor Aeroacoustic Model (TRAM DNW) Performance, Airloads, and Structural Loads," Tech. rep., National Aeronautics and Space Adminstration, 2000.

[64] Lee-Rausch, E. M., and Biedron, R. T., "Simulation of an Isolated Tiltrotor in Hover with an Unstructured Overset-grid RANS Solver," *AHS International 65th Forum and Technology Display*, 2009.

[65] Spalart, P. R., "Comments on the Feasibility of LES for Wings, and on a Hybrid RANS/LES Approach," *Proceedings of First AFOSR International Conference on DNS/LES*, Greyden Press, 1997, pp. 137–147.

[66] Spalart, P. R., Deck, S., Shur, M. L., Squires, K. D., Strelets, M. K., and Travin, A., "A New Version of Detached-eddy Simulation, Resistant to Ambiguous Grid Densities," *Theoretical and Computational Fluid Dynamics*, Vol. 20, No. 3, 2006, pp. 181–195.

[67] Biedron, R., and Thomas, J., "Recent Enhancements to the FUN3D Flow Solver for Moving-Mesh Applications," *AIAA Aerospace Sciences Meeting*, 2009.

[68] Swanson, S., McCluer, M., Yamauchi, G., and Swanson, A., "Airloads Measurements from a 1/4-Scale Tiltrotor Wind Tunnel Test," *25th European Rotorcraft Forum*, September 1999.

[69] Chaderjian, N., "Advances in Rotor Performance and Turbulent Wake Simulation using DES and Adaptive Mesh Refinement," *Proceedings of the Seventh International Conference on Computational Fluid Dynamics*, 2012.

[70] MacLean, M., Mundy, E., Wadhams, T., Holden, M., and Parker, R., "Analysis and Ground test of Aerothermal Effects on Spherical Capsule Geometries," *AIAA Paper 2008-4273*, 2008.

[71] Wright, M. J., Candler, G. V., and Bose, D., "Data-Parallel Line Relaxation Method for the Navier-Stokes Equations," *AIAA Journal*, Vol. 36, No. 9, 1998, pp. 1603–1609. https://doi.org/10.2514/2.586.

[72] Hollis, B. R., and Borrelli, S., "Aerothermodynamics of Blunt Body Entry Vehicles," *Progress in Aerospace Sciences*, Vol. 48, 2012, pp. 42–56.

[73] Nastac, G., Tramel, R., and Nielsen, E. J., "Improved Heat Transfer Prediction for High-Speed Flows over Blunt Bodies using Adaptive Mixed-Element Unstructured Grids," *AIAA Paper 2022-0111*, 2022.

[74] Park, M. A., "Anisotropic Output-Based Adaptation with Tetrahedral Cut Cells for Compressible Flows," Ph.D. thesis, Massachusetts Institute of Technology, Sep. 2008.

[75] Dey, S., Aubry, R. M., Karamete, B. K., and Mestreau, E. L., "Capstone: A Geometry-Centric Platform to Enable Physics-Based Simulation and System Design," *Computing in Science & Engineering*, Vol. 18, No. 1, 2015, pp. 32–39.

[76] McCloud, P. L., "Best Practices for Unstructured Grid Shock Fitting," *AIAA Paper 2017-1149*, 2017.

[77] Lawry, M., and Opgenorth, M., "Parametrically Uniform Mesh Adaptation for Unstructured Grids," *Eleventh International Conference on Computational Fluid Dynamics*, 2022.