Implicit Thermochemical Nonequilibrium Flow Simulations on Unstructured Grids using GPUs

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

Abdelkader Frendi§
University of Alabama in Huntsville, Huntsville, Alabama, 35899

Thermochemical nonequilibrium flow simulation capabilities have been previously implemented, verified, and validated for central processing unit (CPU) systems in NASA's unstructured-grid computational fluid dynamics solver FUN3D. Many exascale-class high-performance computing systems will rely on graphics processing unit (GPU) architectures for high throughput and energy efficiency; thus, CPU-based scientific computing software unable to effectively utilize these systems must be updated. In this work, we present a CUDA C++ implementation of FUN3D's thermochemical nonequilibrium flow simulation capabilities targeting NVIDIA Tesla GPUs. An overview of the porting and optimization strategy is described and performance comparisons with other recent architectures are presented. Scaling to thousands of GPUs is demonstrated, yielding computational performance equivalent to that of several million CPU cores. The implementation enables efficient, high-fidelity, scale-resolving simulations of thermochemical nonequilibrium flows for many applications including atmospheric entry, hypersonics, and combustion.

Nomenclature

$\mathbf{A}$	coefficient matrix
A	Arrhenius factor
C_D	drag coefficient
C_{DES}	DES coefficient
C_p^s	specific heat of species s at constant pressure
C_v^s	specific heat of species s at constant volume
C_p	specific heat at constant pressure
c_{b1}, c_{b2}	turbulence model coefficients
c_{v1}, c_{w1}	turbulence model coefficients
c_{t3}, c_{t4}	turbulence model coefficients
D	diffusivity, also diameter
d	distance to the wall
$\hat{d}$	distance to the wall, optionally modified for DES
E	total energy per unit volume
E_a	activation energy
e	internal energy
$e_{s,o}$	energy of formation of species s at the reference temperature
$\mathbf{F}$	flux tensor
f_{t2}	turbulence model function
f_{v1}, f_{v2}	turbulence model functions
f_w	turbulence model function
FPn	IEEE 754 n-bit floating-point format
GB	gigabyte
g	turbulence model function
H	total enthalpy
h	enthalpy
h_{max}	maximum distance to adjacent dual cell centroid
$h_{s,o}$	enthalpy of formation of species s at the reference temperature
$\mathbf{I}$	identity matrix
J	diffusive flux, also jet-to-crossflow momentum flux ratio
K	equilibrium constant
k	reaction rate coefficient
M	molecular weight, also Mach number
N_{eq}	number of equations
N_r	number of reactions
N_s	number of species
Pr	Prandtl number
p	pressure
$\mathbf{q}$	vector of conserved variables

*Research Scientist, Member AIAA
†Research Scientist
‡Senior Research Scientist, Associate Fellow AIAA
§Professor, Associate Fellow AIAA

$\mathbf{R}$	residual vector
R	reaction rate, also residual vector component and gas constant
Re	Reynolds number
R_n	nose radius
r	turbulence model function
$\mathbf{S}$	source term vector
S	strain rate tensor
$\hat{S}$	turbulence model function
Sc	Schmidt number
T	temperature
TB	terabyte
t	time
U	velocity magnitude
u	Cartesian velocity component
V	dual volume
W	spin tensor
x	Cartesian direction
y	mass fraction
α, β	stoichiometric coefficients
β	Arrhenius temperature exponent
δ	Kronecker delta
Δ	grid spacing
$\dot{q}$	heat flux
Θ	cylinder angle
κ	mixture thermal conductivity, also turbulence model constant
μ	mixture viscosity
σ	turbulence model coefficient
ν	kinematic viscosity
$\tilde{\nu}$	turbulence working variable
ρ	mixture density
τ	shear-stress tensor, also pseudotime
τ_c	convection time
χ	turbulence model function
Ω	vorticity magnitude
$\dot{\omega}$	chemical source term

Subscripts / Superscripts

a	crossflow direction
b	backward
e	energies
f	forward
i	inviscid
j	jet condition
n	wall-normal
ref	reference value
s	species
t	turbulent
u	universal
V	vibrational-electronic-electron
v	viscous
vib	vibrational-electronic
0	quantity at total conditions
∞	quantity at freestream conditions
$\hat{}$	approximate

Symbols

i,j,m,n,r,s	indices

I. Introduction

Computational fluid dynamics (CFD) has become a widely used tool for science and engineering in recent decades. Historically, CFD has primarily relied on Reynolds-averaged Navier-Stokes (RANS) methods which, while computationally efficient, lack accuracy for certain physical phenomena such as flow separation and combustion. As computing power continues to increase, scale-resolving methods such as detached-eddy simulation (DES) [1] and large-eddy simulation (LES) [2] become more computationally feasible for general science and engineering problems. Scale-resolving simulations, which involve a broad range of temporal and spatial scales, generally require many more degrees of freedom than RANS simulations, and critically rely on efficient and scalable flow solvers.

This CFD proliferation is largely due to the exponential increase in computing power of central processing units (CPUs). While CPU performance has grown considerably in recent decades, single-threaded performance has recently stagnated. Hardware designers are now increasing parallelism through the introduction of more processing elements and alternative architectures such as graphics processing units (GPUs). Recently, GPU hardware has accelerated a broad range of applications from deep learning [3] to traditional scientific simulations [4, 5]. As computer architectures offer dramatically more parallelism, algorithms must be adapted to exploit the emerging hardware effectively.

Current plans for U.S. exascale systems [6–8] all rely on GPU acceleration to achieve exascale performance. The NASA CFD Vision 2030 study [9] emphasizes the need for effective utilization of this next generation hardware to achieve the simulation goals of the aerospace community. The NASA Langley Research Center unstructured-grid CFD solver FUN3D [10] has recently been adapted for NVIDIA Tesla GPU architectures [11] and has been used to demonstrate breakthrough performance for perfect gas applications. A GPU implementation is now considered for the generic gas path of FUN3D, which models thermochemical nonequilibrium flows.

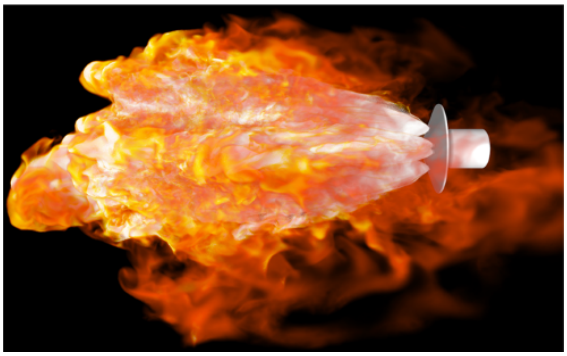


Fig. 1 Human-scale Mars lander concept simulation [11].

Compressible Navier-Stokes solvers for perfect gas flows typically solve five governing equations (conservation of mass, momentum, and energy) and, optionally, auxiliary turbulence model equations. Formulations describing thermochemical nonequilibrium flows add a conservation equation for each chemical species and include multiple energy equations, which leads to systems of significantly greater dimensionality. Through the increased computational cost of additional equations, the modeling of unsteady thermochemical nonequilibrium flows with fine temporal and spatial scales becomes prohibitively expensive; thus, reacting flow simulations have historically relied on RANS methods. Effective use of GPUs enables high-fidelity simulations of reacting flows and opens opportunities for including such simulations earlier in design and development cycles.

An application of interest for NASA is simulation of human-scale Mars atmospheric entry. Such missions require an order of magnitude increase in payload capability, calling for entry, descent, and landing (EDL) concepts relying on complex retropropulsion strategies [12]. CFD simulations of such flows require unsteady general gas chemistry models and are computationally expensive. A single solution typically requires hundreds of thousands of CPU hours with severe limitations on spatial and temporal resolution. FUN3D was recently used to perform perfect gas retropropulsion simulations of a concept vehicle measuring 16 meters in diameter on the GPU-based Summit supercomputer located at the Oak Ridge Leadership Computing Facility [11]. Solution ensembles were performed to investigate the effects of spatial resolution, freestream Mach number, and throttle settings. Spatial domains were measured in kilometers and flow features on the scale of centimeters were resolved as shown in Fig. 1. These campaigns demonstrated that ensembles of simulations which might normally require years on a conventional architecture managed by a capacity policy could instead be completed in several days through the use of a leadership-class GPU-based system operating under a capability policy [11, 13, 14].

The goal of the current work is to extend the GPU-based capabilities of FUN3D to chemically reacting flows. The following sections detail the governing equations for thermochemical nonequilibrium flow, their numerical discretization, the GPU implementation and optimization, verification, and representative results including processor-level performance and efficiency at scale.

II. Governing Equations and Numerical Implementation

A. Governing Equations

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, $\rho = \sum_{s=1}^{N_s} \rho y_s$ 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.

Constitutive relations for pressure, energy, enthalpy, the shear-stress tensor, diffusive flux, heat transfer, and the chemical source term are required to close the equation set. The gas is assumed to be an ideal gas; the pressure is thus defined as

$$p = \sum_{s=1}^{N_s} \rho_s \frac{R_u}{M_s} T = \sum_{s=1}^{N_s} \rho_s R_s T, \tag{4}$$

where T is the temperature, R_u is the universal gas constant, and M_s and R_s are the molecular weight and gas constant of species s, respectively.

The total energy E is defined as

$$E = \frac{1}{2} u_i u_i + \sum_{s=1}^{N_s} y_s e_s, \tag{5}$$

where e_s is the internal energy of species s. The internal energy is assumed to have fully excited rotational and translational modes and is defined as

$$e_s = \int_{T_{\text{ref}}}^{T} C_v^s dT + e_{s,o}, \tag{6}$$

where C_v^s is the specific heat of species s at constant volume, T_{ref} is the reference temperature, and $e_{s,o}$ is the energy of formation of species s at the reference temperature. The total enthalpy is $H = E + p/\rho$. In a similar fashion, species enthalpy h_s is defined as

$$h_s = \int_{T_{\text{ref}}}^{T} C_p^s dT + h_{s,o}, \tag{7}$$

where $C_p^s = C_v^s + R_u/M_s$ is the specific heat of species s at constant pressure and $h_{s,o} = e_{s,o} + R_u T_{\text{ref}}/M_s$ is the enthalpy of formation of species s. NASA polynomials [15] are used to compute these properties on a per-species basis.

Viscous transport is closed with a Newtonian model with turbulence modeled using the Boussinesq eddy viscosity assumption [16]

$$\tau_{ij} = 2\left(\mu + \mu_t\right) S_{ij}, \tag{8}$$

where μ is the dynamic viscosity, μ_t is the turbulent eddy viscosity computed by a turbulence model, and S_{ij} is the strain rate tensor computed as

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}. \tag{9}$$

Diffusive transport is closed with Fick's law

$$J_{sj} = \rho\left(D_s + D_t\right)\frac{\partial y_s}{\partial x_j}, \tag{10}$$

where D_s is the diffusivity of species s and $D_t = \mu_t/(\rho \mathrm{Sc_t})$ is the turbulent diffusivity, where $\mathrm{Sc_t}$ is the turbulent Schmidt number, which is assumed constant.

Heat transfer is closed with Fourier's law

$$\dot{q}_j = \left(\kappa + \kappa_t\right)\frac{\partial T}{\partial x_j}, \tag{11}$$

where κ is the thermal conductivity and $\kappa_t = C_p \mu_t/\mathrm{Pr_t}$ is the turbulent contribution to thermal conductivity, where $\mathrm{Pr_t}$ is the turbulent Prandtl number, which is assumed constant.

The chemical source term of species s per unit volume is defined as

$$\dot{\omega}_s = M_s \sum_{r=1}^{N_r} \left(\beta_{s,r} - \alpha_{s,r}\right) \left(R_{f,r} - R_{b,r}\right), \tag{12}$$

where N_r is the number of reactions, $\alpha_{s,r}$ and $\beta_{s,r}$ are the stoichiometric coefficients for reactants and products in the reaction r, and $R_{f,r}$ and $R_{b,r}$ are the forward and backward reaction rates defined as

$$R_{f,r} = 1000 \left[k_{f,r} \prod_{s=1}^{N_s} \left(0.001 \rho_s / M_s\right)^{\alpha_{s,r}} \right], \tag{13}$$

$$R_{b,r} = 1000 \left[k_{b,r} \prod_{s=1}^{N_s} \left(0.001 \rho_s / M_s\right)^{\beta_{s,r}} \right], \tag{14}$$

where $k_{f,r}$ and $k_{b,r}$ are the forward and backward reaction rate coefficients. The factors are present to convert between commonly used CGS units in reaction rate literature and SI units. The forward reaction rates are closed with the Arrhenius equation with three coefficients

$$k_f = A T^{\beta} \exp\left[-\frac{E_a}{R_u T}\right], \tag{15}$$

where A is the preexponential factor, β is a temperature exponent, and E_a is the activation energy for the reaction. The backward reaction rate is computed based on the equilibrium constant, K, which is generated using Gibbs free energy and fit versus temperature

$$\frac{k_f(T)}{k_b(T)} = K(T). \tag{16}$$

The transport properties (diffusivity, viscosity, and thermal conductivity) are computed using collision integrals [17]. The above set of equations and relations describe general compressible reacting flow in thermal equilibrium. For high-temperature flows such as atmospheric entry, thermochemical nonequilibrium must be considered. For these flows, the two-temperature model [18] may be used where one temperature describes the translational and rotational energies and the second temperature describes the vibrational, electronic, and electron translational energies. The additional transport equation for the vibrational-electronic-electron temperature is

$$\frac{\partial}{\partial t}\left(\rho E_V\right) + \frac{\partial}{\partial x_j}\left(\rho E_V u_j\right) - \frac{\partial}{\partial x_j}\left(\dot{q_V}_j + \sum_{s=1}^{N_s} h_{V\,s} J_{s\,j}\right) = \dot{\omega}_V, \tag{17}$$

where the subscript V denotes vibrational-electronic-electron modes modeled together. Many of the previously described relations now include extra components related to the vibrational-electronic-electron modes. For brevity, the full description of the model is not presented. Further details can be found in [17].

In the computations reported in this paper, turbulence is modeled using the one-equation Spalart-Allmaras model [19] with Catris-Aupoix compressibility corrections [20]:

$$\begin{aligned} \frac{\partial \rho\tilde{\nu}}{\partial t} + u_j \frac{\partial \rho\tilde{\nu}}{\partial x_j} = c_{b1}(1 - f_{t2})\hat{S}\rho\tilde{\nu} - \left[c_{w1} f_w - \frac{c_{b1}}{\kappa^2} f_{t2}\right] \rho \left(\frac{\tilde{\nu}}{\hat{d}}\right)^2 + \\ \frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\mu \frac{\partial \tilde{\nu}}{\partial x_j}\right) + \frac{1}{\sigma}\frac{\partial}{\partial x_j}\left(\sqrt{\rho}\tilde{\nu}\frac{\partial \sqrt{\rho}\tilde{\nu}}{\partial x_j}\right) + \frac{c_{b2}}{\sigma}\frac{\partial \sqrt{\rho}\tilde{\nu}}{\partial x_i}\frac{\partial \sqrt{\rho}\tilde{\nu}}{\partial x_i}. \end{aligned} \tag{18}$$

The turbulent eddy viscosity is computed as

$$\mu_t = \rho\tilde{\nu} f_{v1}, \tag{19}$$

where

$$f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, \tag{20}$$

$\chi = \tilde{\nu}/\nu$ is the ratio of the turbulence variable $\tilde{\nu}$, and the molecular kinematic viscosity, $\nu = \mu/\rho$. $\hat{S}$ is given by

$$\hat{S} = \Omega + \frac{\tilde{\nu}}{\kappa^2 \hat{d}^2} f_{v2}, \tag{21}$$

where $\Omega = \sqrt{2W_{ij}W_{ij}}$ is the magnitude of the vorticity, $\hat{d} = \min(d, C_{DES}h_{max})$ is the minimum of the distance to the wall, d, and the maximum distance to an adjacent dual cell centroid, h_{max}, multiplied by C_{DES} for conventional DES [1]. Table 1 details the standard constants used, and the remaining functions are defined as

$$f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}}, \tag{22}$$

$$f_w = g\left[\frac{1 + c_{w3}^6}{g^6 + c_{w3}^6}\right]^{1/6}, \tag{23}$$

$$g = r + c_{w2}(r^6 - r), \tag{24}$$

$$r = \min\left(\frac{\tilde{\nu}}{\tilde{S}\kappa^2 \hat{d}^2}, 10\right), \tag{25}$$

$$f_{t2} = c_{t3} \exp(-c_{t4}\chi^2), \tag{26}$$

$$c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1 + c_{b2}}{\sigma}, \tag{27}$$

and

$$W_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i}\right). \tag{28}$$

Table 1 Values of turbulence model constants.

Variable	c_{b1}	c_{b2}	σ	κ	c_{w2}	c_{w3}	c_{v1}	c_{t3}	c_{t4}	C_{DES}
Value	0.1355	0.622	2/3	0.41	0.3	2	7.1	1.2	0.5	0.65

B. Numerical Implementation

FUN3D is CFD software developed at the NASA Langley Research Center and is used to simulate complex flows across the speed range [10]. The equations are solved with the method of lines (MOL). The spatial domain is discretized using a node-based finite-volume approach on general unstructured grids that can include tetrahedra, pyramids, prisms, and hexahedra. A median-dual volume is constructed around each node by connecting the edge midpoints with the face centers and the centroids of the primal elements. Fig. 2 illustrates the dual and primal cell volumes in two dimensions.

The inviscid fluxes are computed at edge medians using an approximate Riemann solver such as Roe's scheme [21]. Second-order accuracy is obtained using an unstructured Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) reconstruction [22, 23] with unweighted least-squares gradients computed at each node. The viscous fluxes are discretized using Green-Gauss element-based gradients. For tetrahedral grids, this is equivalent to a Galerkin-type approximation. On other element types, the edge-based gradients are combined with the Green-Gauss element-based gradients; this improves the stability of the viscous operator. The equations are integrated in time using implicit formulations such as first-order backward Euler or second-order backward difference (BDF2) approximations to the time derivative. The conservation equations can be rewritten in vector form

$$\int_V \frac{\partial \mathbf{q}}{\partial t} dV + \oint_S (\mathbf{F} \cdot \mathbf{n})\, dS - \int_V \mathbf{S} dV = \mathbf{0}. \tag{29}$$

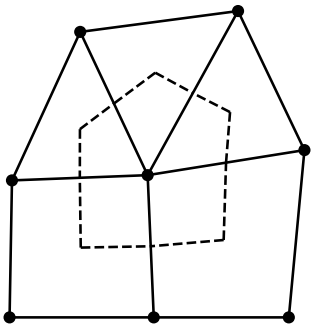


Fig. 2 Grid nomenclature: Solid lines denote primal grid elements while dual volumes are denoted by dashed lines.

The total number of equations is $N_{eq} = 3 + N_s + N_e + N_t$, where three equations represent momentum conservation, N_s is the number of species, N_e is the number of energies, and N_t is the number of turbulence model equations. The vector $\mathbf{q} = [\rho\vec{y}_s, \rho\vec{u}, \rho E, \rho E_V, \rho\tilde{\nu}]^T$ is the vector of conserved variables, $\mathbf{F} = \mathbf{F}_i - \mathbf{F}_v$ is the $N_{eq} \times 3$ flux tensor, $\mathbf{F}_i$ is the inviscid flux tensor, $\mathbf{F}_v$ is the viscous flux tensor, and $\mathbf{S}$ is the source term vector. Integration is performed over the dual volumes.

The discrete nonlinear equations can be written in a compact form where time integration is taken as first-order backward Euler for simplicity

$$V\frac{\mathbf{q}^{n+1} - \mathbf{q}^n}{\Delta t} + \mathbf{R}(\mathbf{q}^{n+1}) = \mathbf{0}. \tag{30}$$

Here, V is the dual volume, Δt is the physical time step, $\mathbf{q}^n$ is the vector of conserved variables defined at time step n, and $\mathbf{R}$ is the spatial residual vector. The system of nonlinear equations is solved using a defect correction method and pseudotime as

$$\left[\frac{V}{\Delta\tau}\mathbf{I} + \frac{V}{\Delta t}\mathbf{I} + \frac{\partial\hat{\mathbf{R}}}{\partial\mathbf{q}}\right]\Delta\mathbf{q} = -\mathbf{R}(\mathbf{q}^{n+1,m}) - \frac{V}{\Delta t}\left(\mathbf{q}^{n+1,m} - \mathbf{q}^n\right), \tag{31}$$

$$\mathbf{q}^{n+1,m+1} = \mathbf{q}^{n+1,m} + \Delta\mathbf{q}. \tag{32}$$

Here, m is the implicit nonlinear iteration index, $\Delta\tau$ is the local pseudotime step, $\mathbf{I}$ is the identity matrix, and $\mathbf{q}^{n+1,1} = \mathbf{q}^n$. After completion of nonlinear iterations, the final solution at $n+1$ is denoted as $\mathbf{q}^{n+1}$. The approximate Jacobian $\partial\hat{\mathbf{R}}/\partial\mathbf{q}$ is based on a linearization of first-order inviscid fluxes and second-order viscous fluxes. The linearization is performed around the latest solution, $\mathbf{q}^{n+1,m}$, and involves only contributions from nearest neighbors. The Jacobian matrix is sparse and composed of $N_{eq} \times N_{eq}$ dense blocks. The matrix $\mathbf{A} = \left[\frac{V}{\Delta\tau}\mathbf{I} + \frac{V}{\Delta t}\mathbf{I} + \frac{\partial\hat{\mathbf{R}}}{\partial\mathbf{q}}\right]$ is segregated into diagonal and off-diagonal blocks. The number of off-diagonal blocks in each row corresponds to the number of nearest neighbors and is typically between 15 (regular tetrahedral grids) and 26 (regular hexahedral grids).

A multicolor point-implicit approach is used to obtain approximate solutions of the linear system of equations at each nonlinear iteration. In this approach, the system of equations is colored such that no two adjacent unknowns are assigned the same color. Colors are processed sequentially. 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. Unknowns within each color are further segregated into those comprising interior values and those required for partial halo exchanges with neighboring grid partitions in order to facilitate effective overlapping of communication and computation while recovering the serial algorithm. Finally, the entire system of equations is renumbered to achieve optimal use of memory bandwidth. Since the discrete nonlinear iterations are cast in update form, judicious use of mixed-precision arithmetic is leveraged to reduce memory traffic. The GPU implementation was extensively optimized for a broad range of matrix block sizes in previous work and supports FP64, FP32, and FP16 arithmetic. For further details on the implementation, the reader is referred to Refs. [24] and [25].

III. GPU Implementation

FUN3D was originally a CPU-based code written in modern Fortran. Coarse-grained message passing interface (MPI) [26] with conventional domain decomposition techniques are used for parallelism, although a hybrid MPI+OpenMP [27] implementation is also partially supported. Viewed from a high level, FUN3D is organized as a series of subroutine calls that transform mutable data structures (examples include the residual vector and Jacobian matrix appearing in Eq. 31). Here, these subroutines are referred to as compute kernels. FUN3D is designed such that any library of computational kernels may be leveraged, provided they conform to a standard C99 interface. To utilize GPU hardware, an external library called FLUDA has been developed. The following sections detail the programming model and target hardware, design approach, porting procedure, and optimization of GPU-based FUN3D. Here, the expressions *host* and *device* serve as general terms referring to CPU and GPU processors, respectively.

A. Programming Model and Hardware Environment

NVIDIA GPUs utilize single instruction, multiple data (SIMD) execution and are thus optimal for computations that can be run concurrently on multiple data elements. Computations may be divided over thousands of threads, which are processed using streaming multiprocessors (SMs). The threads are organized into blocks, which are themselves organized into warps (currently 32 threads) that run on SMs.

Based on prior investigations into available programming models, CUDA C++ [28] (henceforth CUDA) is the approach currently used to target NVIDIA GPU hardware. The library used by FUN3D to solve perfect and generic gas flows on such devices is known as FLUDA. FLUDA is a port of a subset of compute and auxiliary kernels from FUN3D's baseline Fortran implementation. The implementation does not require any external libraries. Since the legacy Fortran kernels are tightly integrated within FUN3D, the use of an external implementation such as FLUDA is controlled by a runtime input parameter. Fig. 3 shows an example of a typical kernel structure. Combining FUN3D and FLUDA, the result is a single executable capable of running a simulation on CPUs or GPUs as specified at runtime. In the current implementation, supported hosts include x86, IBM POWER, or ARM CPU processors. Valid devices include NVIDIA Tesla P100, V100, or A100 GPUs.

```
subroutine fortran_kernel(args)
  use cuda_vars, only: use_cuda, cuda_args
  use fluda_interface, only: fluda_interface_kernel
  ! argument definitions
  continue
  if (use_cuda) then
    call fluda_interface_kernel(args,cuda_args)
    return
  end if
  ! CPU path
end subroutine fortran_kernel
```

Fig. 3 FLUDA kernel call in FUN3D.

B. Design Approach

In the current implementation, all input parsing, domain decomposition, and other initialization functions are performed on the host as in the conventional CPU-based paradigm. While these operations could potentially be performed on the GPU, the mapping of such kernels to a GPU architecture would require substantial effort, yet would likely realize a smaller performance benefit. Furthermore, the cost of these operations is typically amortized over a complete simulation, particularly for long-duration unsteady applications. Once preprocessing operations are complete, the data required to run a solution are moved to the GPU and remain there for the duration of the simulation.

Due to the high latencies associated with moving data between the host and device, minimizing the need for data motion during the solution process is of paramount importance. Moreover, Amdahl's Law requires that all compute kernels execute on the device. To satisfy these key requirements, all kernels required for the integration of the governing equations have been migrated to the GPU. With the exception of optional host-based MPI buffers as described later in this section, the only data requiring transfer between the host and device during the solution process are scalars related to solution monitoring such as residual norms and engineering outputs. Note that in this context, the role of the CPU is solely high-level control flow and scheduling the launch of CUDA kernels on the GPU; the CPU is not used for any computations during the solution of the governing equations.

Large quantities of output data, such as those needed to support visualization, are optional and only transferred to CPU memory at user-specified intervals. The implementation also supports efficient output of time-varying volumetric solution data to external storage devices. Solution data is copied from the GPU to asynchronous Fortran I/O buffers on the CPU. These buffers are then written to external storage in an asynchronous fashion. This approach has been effectively used to export hundreds of terabytes of solution data during the course of a simulation with minimal impact on computational efficiency [13].

By design, all data structures are identical between the CUDA and Fortran contexts. Multidimensional array layouts are kept in column-major order to be consistent with Fortran and to avoid the need for expensive transformations. For more complex data structures beyond the primitive data types, interoperable "mirror" types are created for use in CUDA (described in more detail below). The implementation is intended to maintain discrete consistency with the CPU solution procedure. All variables that are FP64 in the CPU implementation are also FP64 in the GPU implementation. By default, and for the simulations included here, the off-diagonal matrix coefficients and update vector appearing in Eqs. 31 and 32 are stored as FP32 as in the CPU implementation. These data structures can be optionally stored as FP64 with a compile-time flag at the cost of additional execution time and memory.

The default coarse-grained parallel execution paradigm for multinode and/or multi-GPU systems is to bind a single MPI rank to each GPU. In this context, many of the processing cores available on a typical host processor remain idle during execution. Code has been developed to automate device selection for nodes containing multiple GPUs and to manage the CUDA Multi-Process Service (MPS) [29] when multiple MPI ranks per GPU are desired. This approach permits the effective use of CPU resources for kernels which remain on the host, such as those associated with preprocessing or optional kernels such as boundary and feature extracts, slicing, or other visualization artifacts.

The FUN3D MPI interface has been extended to support the use of device pointers for environments in which the MPI implementation supports CUDA-aware MPI calls. If conventional host-based MPI buffers are used, they are page-locked using `cudaHostRegister` to facilitate fast transfers. If the data moved between the host and device are minimized, the cost of such transfers is generally negligible and application performance scales well to leadership-class systems with thousands of GPUs. Without a specific software stack and tuned protocol parameters, the host-based MPI paradigm generally outperforms CUDA-aware MPI/GPUDirect for FUN3D communication, which is typically characterized by many small messages comprising partial halo exchanges within the multicolor linear solver algorithm described in Sec. II.B.

C. GPU Porting Strategy

A mini-application, or *mini-app* infrastructure [30] is employed to port Fortran compute kernels to CUDA. A mini-app approach facilitates rapid prototyping and optimization of individual kernels, avoiding cumbersome development issues associated with the full application infrastructure consisting of nearly one million lines of code and its requisite build system. The mini-app structure and workflow is shown schematically in Fig. 4 and is designed to replicate the computations associated with a single time step given by Eqs. 31 and 32.

A small FUN3D simulation is first performed to generate a snapshot of the entire application state at some instant in time which is then stored to disk. The mini-app execution begins by loading this state into an internal data module and constructing a mirror of the CPU state variables in device memory. A series of C99 interfaces which can be called from Fortran using its `iso_c_binding` module are used to wrap device memory functions such as `cudaMemcpy`. In general, the primitive CPU state variables translate directly to CUDA and need only be copied to the device and stored as device pointers using the `iso_c_binding c_ptr` type.

To facilitate semantically identical access to arrays of Fortran derived types containing allocatable arrays in the CUDA context, an interoperable mirror type with the `bind(c)` attribute is created, wherein allocatable arrays instead appear as type `c_ptr`. The entire interoperable array of derived types can then be copied directly to the device and accessed using a C99 `struct` variable with the same order of member variables. See Fig. 5 for an example of these types.

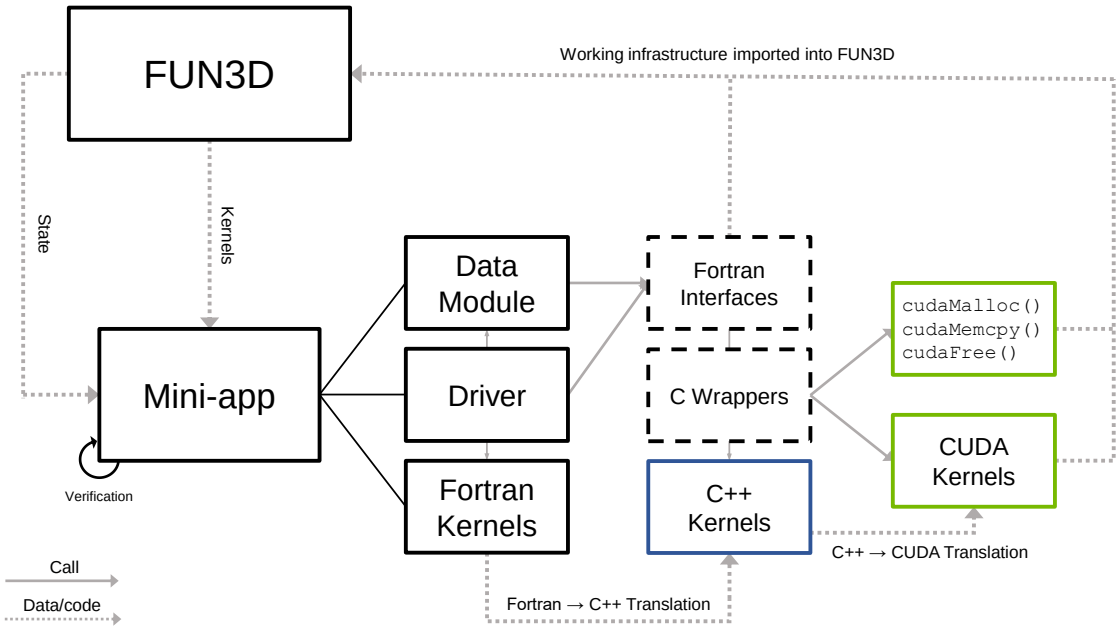


Fig. 4 FUN3D mini-app structure and porting workflow.

Original Derived Type

```
type, public :: elem_type
  integer :: a
  integer, dimension(:,:),&
    allocatable :: b
  integer, dimension(:,:,:),&
    allocatable :: c
end type elem_type
```

Interoperable Mirror Derived Type

```
type, bind(c) :: elem_type_p
  integer(c_int) :: a
  type(c_ptr) :: b
  type(c_ptr) :: c
end type elem_type_p
```

Interoperable C Struct

```
typedef struct elem_type_p {
  int a;
  int* b;
  int* c;
} elem_type_p;
```

Fig. 5 Interoperable mirror types for CUDA.

With the data module and mirror data in place, the kernel porting process begins by developing C99 wrappers for each CUDA kernel which are callable from Fortran. Scalar arguments and device pointers are passed to these kernels from the mini-app driver module. Each Fortran compute kernel is then translated into an equivalent CUDA kernel in a naive (verbatim, where possible) manner. An optional intermediate step is to translate the kernel to C++ CPU code to verify correctness before incorporating a GPU threading strategy. Note that kernel optimizations are not pursued at this stage.

In the CPU implementation, all operations are assumed to occur serially within each MPI rank. Therefore, gather-scatter operations typically associated with the assembly of flux and Jacobian contributions during an unstructured-grid traversal do not introduce race conditions. However, when a shared-memory programming model such as OpenMP or CUDA is introduced, threads must be prevented from simultaneously updating the same memory location. While this issue may be addressed at the application level through strategies such as coloring techniques, NVIDIA GPUs provide convenient hardware support for atomic operations in global memory. This approach is used in the current implementation and is found to be the most straightforward and performant strategy.

Approximately 40 new CUDA kernels have been developed to support the generic gas capability in addition to the existing 130 kernels previously established for perfect gas simulations. The kernels vary in length from a few lines to thousands of lines. After the kernels have been ported and verified, FLUDA is integrated into FUN3D. At the beginning of each Fortran kernel, device pointers initialized during preprocessing are provided through the module interface. If the user has requested GPU execution, the CUDA kernel is scheduled for execution in the default CUDA stream and control returns to the CPU as indicated in Fig. 3.

1. Verification and Floating-Point Considerations

Kernel verification is performed by comparing kernel outputs with those of the corresponding Fortran kernel. The relative difference between outputs should not exceed some specified tolerance, e.g., 1×10^{-14} and 1×10^{-7} for FP64 and FP32, respectively. In general, the RMS norm of the differences between two output arrays will be within the tolerance if both implementations are correct, although several values may exceed the tolerance for a number of reasons.

FP64 values generally do not match to machine precision due to order-of-operations differences between the serial CPU and massively parallel GPU, since floating point operations are not necessarily associative. This effect may be mitigated during testing by running a single thread on the GPU; however, this assumes the compilers and runtime produce the same orderings and that the GPU code has not been altered nor optimized, which is rarely the case. The situation may be further exacerbated by fused multiply-add operations which may be unavailable (or unused) on the CPU, though these may be disabled on the GPU for testing. Catastrophic cancellation is also a common source of differences between CPU and GPU results. Finally, code branches which are executed based on a numerical threshold may result in a value for the CPU that falls on one side of the threshold and a value for the GPU that falls on the other side of the threshold due to order-of-operations differences.

The differences noted here are an inevitable consequence of finite-precision arithmetic and not a problem specific to GPUs per se. Serial CPU results, though repeatable, are no more correct than GPU results which differ with each asynchronous execution. If the ordering of loops is arbitrarily permuted for each CPU execution, differences commensurate with those produced by the GPU are observed.

D. Kernel Optimization

Several fundamental tenets of GPU programming have been used to guide kernel optimizations in the current implementation. GPUs derive their performance from massive concurrency that requires computations be parallelized over thousands of thread blocks in order to effectively hide latencies. A simple parallelization over grid entities, such as edges or elements, may be insufficient to saturate the available GPU resources, requiring additional avenues of parallelism to be exposed. Since GPU threads are extremely lightweight in comparison with CPU threads, the resources allocated to each GPU thread should be minimal. Coalesced memory accesses are critical, and warps consisting of 32 threads may suffer severe performance penalties if any two threads execute divergent conditional branches.

1. Reducing Kernel State

In many CUDA kernels, minimizing the number of temporary variables is necessary to alleviate register pressure. For perfect gas kernels where the number of governing equations for the mean flow is always five, many of the necessary workspace array sizes can be hard-coded. In the generic gas context where the number of equations may vary, the Fortran implementation relies on numerous variable-length arrays (VLAs) as workspace. VLAs are used to such an extent that a naive CUDA port results in stack frames on the GPU so large that the device will immediately run out of memory on cases of interest. These arrays are typically computed in loops, often over the number of equations. The amount of concurrency can be increased by assigning these loop iterations to GPU threads. This technique can eliminate many of the arrays outright. Those that cannot be eliminated are often shared by the threads and so may be placed in shared memory, which alleviates the performance penalty.

Since VLAs are not supported in CUDA, templating is extensively used. The three templated variables are the number of species, energy equations, and turbulence variables. Because these variables are not known at compile time, X-Macros are used for all kernels that require templated variables. In this approach, a range of combinations of these templated variables is specified at compile time for use at run time. An example kernel call is shown in Fig. 6 with an example list definition shown for reference, typically stored in a separate file.

```
#define SPECIES_ENERGY_TURB_LIST \
  X(1,1,0) \
  X(5,1,0) \
  X(5,2,0) \
  X(1,1,1) \
  X(5,1,1) \
  X(5,2,1)

#define X(n_s,n_e,n_t) \
  if (n_species == n_s && n_energy == n_e && n_turb == n_t) \
  cuda_kernel<n_s,n_e,n_t><<<nBlocks, nThreads>>>(ARGS);
  SPECIES_ENERGY_TURB_LIST
#undef X
```

Fig. 6 X-Macro function call for an example kernel.

2. Loop Flattening and Thread Divergence

Concurrency may also be increased through loop flattening. In a typical FLUDA compute kernel, a block of threads is two-dimensional and X-dimension threads work together to compute one work item. X-dimension threads loop in `blockDim.x`-stride chunks over the total iterations of the flattened loop. Data computed in the outer loops are precomputed and stored in shared memory to ensure that the main work loop threads encounter as little branching and latency as possible. An example of this type of kernel appears in Fig. 7. Fig. 7a shows an example of where loop splitting would help reduce thread divergence. As shown in Fig. 7b, the loop could be split into two, where k varies first from 0 to $S-1$ and then S to $S+3$.

```
for(int ii=threadIdx.x,last=N*(S+4);ii < last;ii+=BLOCK_DIM_X) {
  int j = ii / (S+4);
  int k = ii % (S+4);
  int ji = MAX(((j%Z)-S+1),0);

  if (k < S) {
    atomicAdd( &AD(k,j,idl), sign[ji]*k_fac[k]*fac1[j] );
  } else {
     atomicAdd( &AD(k,j,idl), m2*sign[ji]*fac1[j] );
  }
}
```

(a)

```
for(int ii=threadIdx.x,last=N*S;ii < last;ii+=BLOCK_DIM_X) {
  int j = ii / S;
  int k = ii % S;
  int ji = MAX(((j%Z)-S+1),0);

  atomicAdd( &AD(k,j,idl), sign[ji]*k_fac[k]*fac1[j] );
}

for(int ii=threadIdx.x,last=N*4;ii < last;ii+=BLOCK_DIM_X) {
  int j = ii / 4;
  int k = ii % 4 + S;
  int ji = MAX(((j%Z)-S+1),0);

  atomicAdd( &AD(k,j,idl), m2*sign[ji]*fac1[j] );
}
```

(b)

Fig. 7 (a) A flattened FLUDA-style work loop. (b) Flattened and split FLUDA-style work loop to decrease divergence.

3. Irregular Memory Accesses

When warp-level memory access patterns are irregular, the performance penalty may be alleviated by loading contiguous segments from main memory into shared memory, which can be used as a workspace and suffers no penalty for noncontiguous accesses. In the most expensive generic gas kernel, memory access and computation are both regularized by computing and storing the transpose of the matrix, which is transposed again upon kernel completion.

4. Kernel Launch Strategies

CUDA kernel performance may vary drastically with block and thread launch parameters. This is especially true for kernels with large work loops, such as the Jacobian matrix construction kernels for high numbers of equations, which may require hundreds of threads per block to achieve optimal performance. This is further exacerbated by the fact that each kernel should run optimally for every possible combination of chemical species, energy equations, and turbulence equations. FLUDA uses an automated brute-force tuning approach to achieve optimal performance across this parameter space. In this approach, FLUDA is compiled using $O(100)$ different threading strategies and kernels are timed for each combination of chemical species, energy equations, and turbulence equations. For each combination of kernel and equations, the best result is chosen as the threading strategy.

Kernels associated with boundary conditions generally represent a small amount of work per Fortran compute kernel, as the routine is called on each individual boundary in legacy FUN3D. To improve saturation of the GPU, a single kernel is instead launched that performs the computation for all appropriate boundaries. This approach may significantly improve performance for cases containing hundreds of boundary instances.

5. Host Updates

As mentioned in Sec. III.B, updates to host data at each time step are generally restricted to residual norms and scalars associated with engineering outputs. Residual norms are stored in an array dimensioned by the number of governing equations that can be efficiently transferred with a single call to `cudaMemcpy`. However, collections of engineering outputs such as pressure and viscous contributions to forces and moments on boundary groups are stored in arrays of derived types with allocatable member arrays. As the approach in Sec. III.C describes, each allocatable array would be represented by a pointer allocated with a `cudaMalloc` call. This implies that separate calls to `cudaMemcpy` would be required to transfer each array. For cases with hundreds of boundary instances, the limitations of this approach quickly become apparent. To mitigate this, a single array is allocated on the host (with a device mirror) and the boundary arrays become pointers into this array. The array, which contains data for all boundaries, can then be transferred with a single call to `cudaMemcpy`.

IV. Results

Four test cases are presented. The first case is a two-dimensional hypersonic flow around a cylinder that is used to verify the GPU implementation. A representative three-dimensional flow around a blunt body is then examined and computational performance is benchmarked on various architectures. The third case is a DES of a transverse reacting hydrogen jet in supersonic cross flow. This case features a transient turbulent flow and utilizes significant computational resources. Lastly, the scalability of the implementation is established on Summit.

A. High Enthalpy Hypersonic Flow around a Cylinder

A high enthalpy hypersonic ($M_\infty \approx 8.7$) flow around a cylinder was experimentally studied at the high enthalpy shock tunnel Göttingen [31]. This case was used for most of the initial conversion of the generic gas kernels due to its simplicity and relatively small problem size. The setup of the case is a free piston shock tunnel. The cylinder radius is 45 mm and the length is 380 mm. Surface heat transfer and pressure are compared to experimental measurements. Specifically, run 627 is considered, where the test gas is air with stagnation values of pressure and temperature equal to $p_0 = 48.3$ MPa and $T_0 = 7369$ K, respectively.

A structured hexahedral grid is used with 181, 181, and 2 nodes in the radial, azimuthal, and axial directions, respectively. The first wall cell Reynolds number, $Re_{\Delta n} = \rho_\infty u_\infty \Delta n/\mu_\infty$, is unity. Air is modeled using a five species (N_2, O_2, NO, N, O), five reaction, mechanism [32]. A two-temperature model is utilized. The freestream mass fractions are set to

$$\vec{y}_\infty = \left[y_{N_2}, y_{O_2}, y_{NO}, y_N, y_O\right]^T = [0.7356, 0.1340, 0.0509, 0.0000, 0.0795]^T \tag{33}$$

to match conditions described in literature [33], for which solutions have been computed based on the plenum reservoir and nozzle geometry. The freestream velocity, temperature, and density are 4776 m/s, 694 K, and 0.003255 kg/m^3, respectively. A laminar flow is assumed due to the freestream Reynolds number. The wall is modeled as noncatalytic with a constant temperature of 300 K. Farfield is imposed on the inflow plane and extrapolation is imposed on the outflow plane. Symmetry is imposed on the axial planes. The equations are integrated in time using the backward Euler approximation to the time derivative and local time stepping with a maximum Courant-Friedrichs-Lewy (CFL) number of 5.0. A total of 80,000 iterations were run with the first 5,000 using first-order spatial accuracy for the inviscid fluxes.

Results computed using a dual-socket CPU and a GPU are presented in Figs. 8-10 and Table 2. Contours based on the GPU simulation included as Fig. 8 show flow quantities of interest. The shock is captured within a few cells of the structured grid. A thin reacting layer of nitric oxide occurs due to the high temperature in the shock. Simulation results computed on the GPU agree very well with the experimental data (Fig. 9) and with previous CPU-based computational studies reported in the literature [33]. At convergence, the drag coefficients for each of the two simulations match to 5.5×10^{-13}, as shown in Table 2. The convergence histories for all equations also agree favorably as can be seen in Fig. 10.

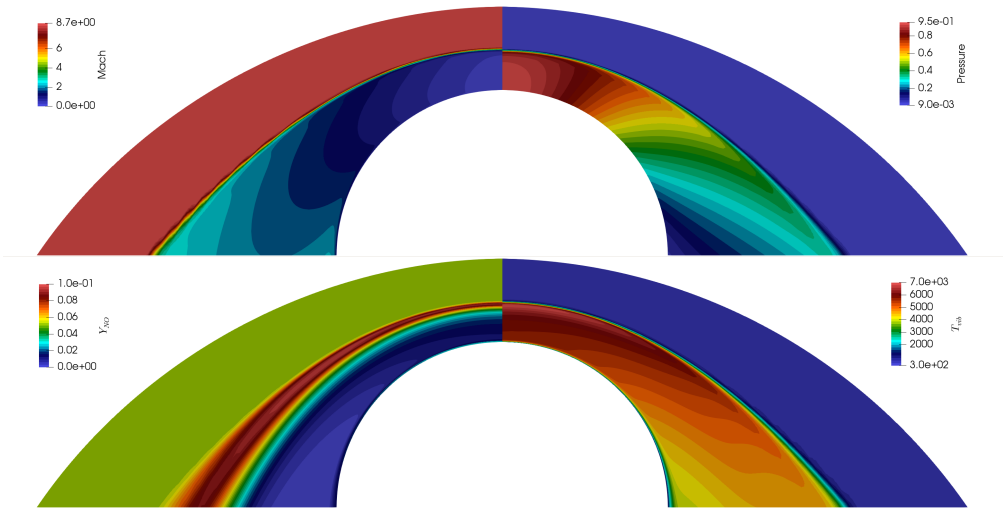


Fig. 8 GPU solution contours for high enthalpy hypersonic flow around a cylinder. Top Left: Mach Number. Top Right: Nondimensional pressure. Bottom Left: Nitric oxide mass fraction (y_{NO}). Bottom Right: Vibrational-Electronic temperature in Kelvin (T_{vib}).

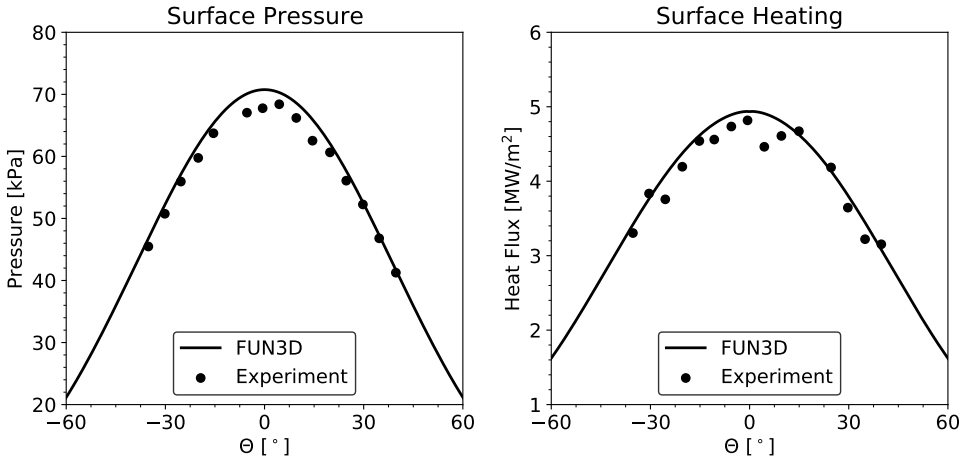


Fig. 9 GPU surface pressure and heat flux with comparison to experiment [31]. Θ is the angle in degrees around the cylinder, centered about the stagnation point.

Table 2 CPU and GPU drag comparison for high enthalpy hypersonic flow around a cylinder.

	C_D
CPU	$4.3791550447125\times10^{-2}$
GPU	$4.3791550447\mathbf{01}\times10^{-2}$
Relative Difference	5.5×10^{-13}

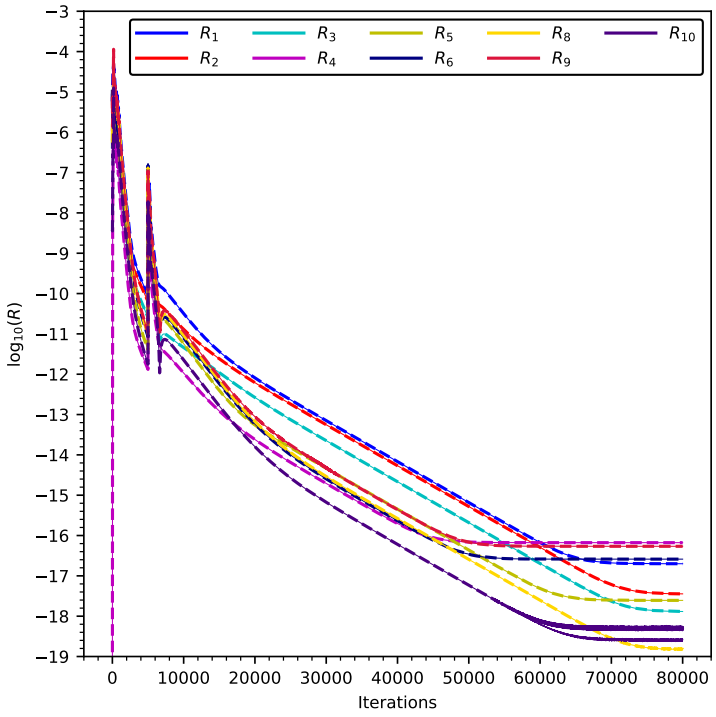


Fig. 10 Residual convergence histories for hypersonic flow around a cylinder, where CPU residuals are denoted by solid lines and GPU residuals are denoted by dashed lines. The y-momentum residual (R7) is exactly zero due to symmetry and not shown.

B. Hypersonic Flow around a Hemisphere Cylinder

A hypersonic flow ($M_\infty \approx 9.8$) around a hemisphere cylinder, which is a representative blunt body, has previously been analyzed experimentally and computationally [34]. The experimental setup utilizes a shock tunnel to produce hypersonic thermochemical nonequilibrium flow in the test section. Surface heat transfer and pressure measurements are available for comparison. Surface heat transfer is computed through two methods: one utilizing coaxial thermocouple gauges and another utilizing thin film gauges. For this example, model two is examined. The radius of the nose, R_n, is one inch and the full length of the body is eight inches.

A general unstructured mixed-element grid is generated for this configuration. The grid is composed of 2,426,333 nodes and 9,153,133 cells (6,556,423 tetrahedra and 2,596,710 prisms). The surface grid is composed solely of triangles. The hemisphere surface spacing is set to $\Delta = 0.04R_n$. The cylinder uses a surface spacing of $\Delta = 0.08R_n$. The prismatic boundary layer grid is composed of 80 layers with a growth rate of 1.1. The first wall cell Reynolds number is approximately 0.03 ($\Delta n \approx 10^{-7}$ m). The remaining volume is meshed with tetrahedra where a volume source with spacing $\Delta = 0.08R_n$ is imposed. A crinkle-cut slice through the centerline is shown in Fig. 11.

Air is modeled using a five species (N_2,O_2,NO,N,O), five reaction mechanism [32]. A two-temperature model is utilized. The freestream mass fractions are set to standard air conditions. The freestream Mach number is 9.8. The freestream temperature and pressure are $T_\infty = 450$ K and $p_\infty = 230$ Pa, respectively. A laminar flow is assumed due to the freestream Reynolds number. The wall is modeled as noncatalytic with a constant temperature of 555.5 K (1000 °R) to match the experimental conditions. Farfield is imposed on the inflow plane and extrapolation is imposed on the outflow plane. The equations are integrated in time using backward Euler with local time stepping with a maximum CFL number of 5.0. Overall, 4,000 iterations have been performed, with the first 2,000 performed with first-order spatial accuracy for the inviscid fluxes.

Results computed using a dual-socket CPU and a GPU are presented in Figs. 12-14 and Table 3. Fig. 12 depicts contours of various solution quantities. Hemisphere surface contours of pressure and heating are smooth due to the bow shock being cleanly captured in the prismatic boundary layer grid. The translational-rotational temperature reaches approximately 8100 K, causing significant dissociation of oxygen and nitrogen leading to production of nitric oxide

near the hemisphere surface. Fig. 13 depicts pressure and heat flux along the centerline of the hemisphere cylinder. The GPU-based solution agrees very well with the computational results reported in literature [34]. While Ref. [34] does not present the uncertainty of the experiment, the authors indicate that the reported computational results are within the scatter of the experimental measurements. At the end of the simulation, the drag coefficients between CPU and GPU match up to six significant digits (Table 3). Residuals also agree favorably (see Fig. 14). As explained in Sec. III.C, agreement to machine precision is not expected until the solution is fully converged due to order-of-operations differences between the CPU and GPU.

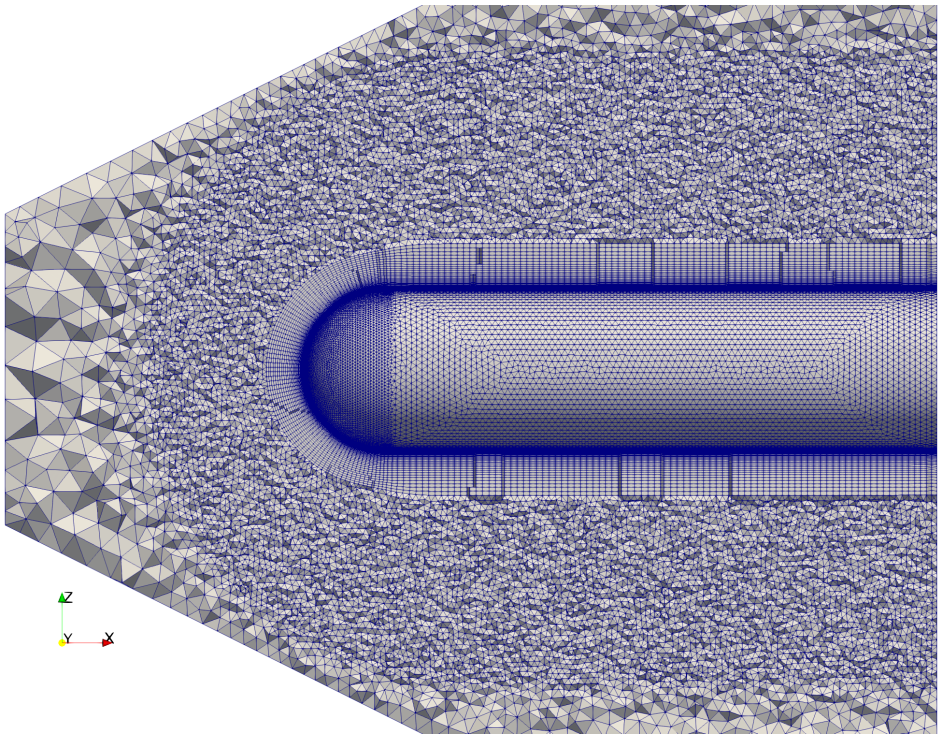


Fig. 11 Hemisphere cylinder surface and volume grid.

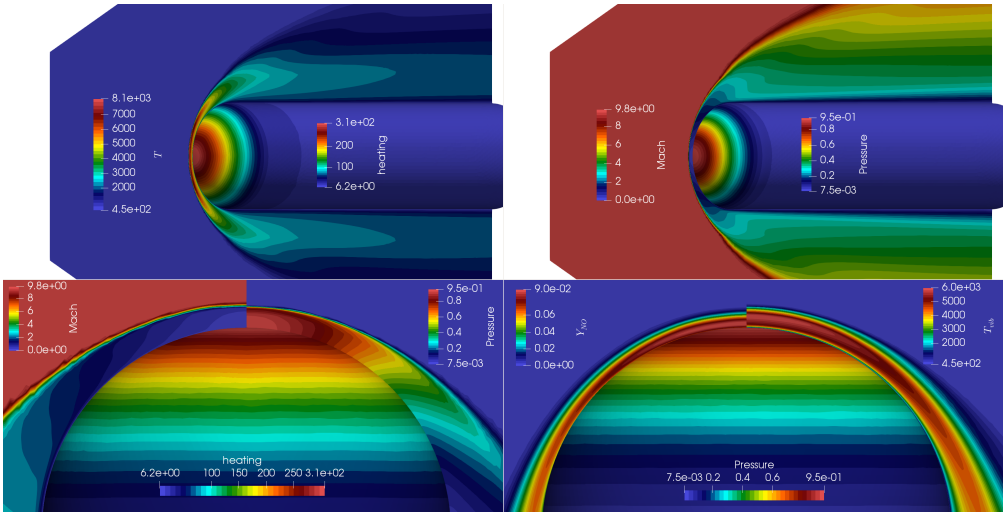


Fig. 12 GPU solution contours for hypersonic flow around a hemisphere cylinder. Contours are plotted for Mach number, nondimensional pressure, temperature (T) in Kelvin, surface heating in W/cm^2, nitric oxide mass fraction (y_{NO}), and vibrational-electronic temperature (T_{vib}) in Kelvin.

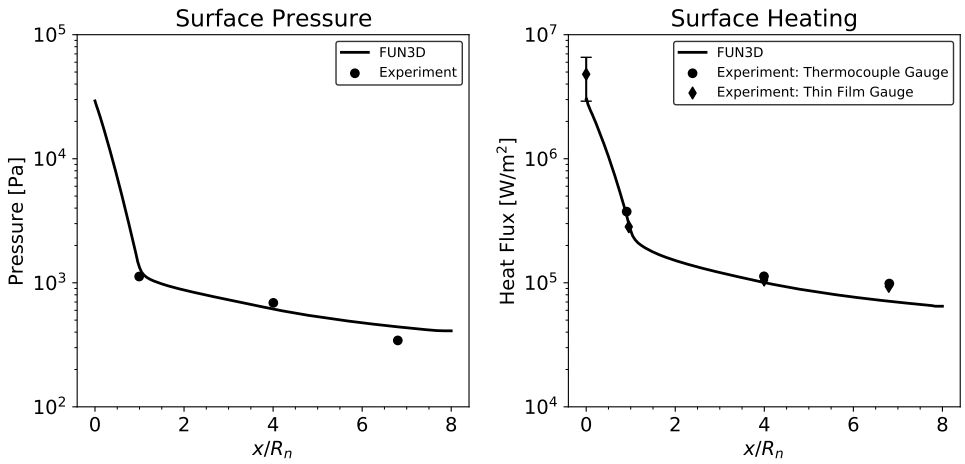


Fig. 13 GPU surface pressure and heat flux with comparison to experiment (obtained from [34]).

Table 3 Hemisphere cylinder CPU and GPU drag comparison.

	C_D
CPU	1.973573461×10^{-3}
GPU	$1.97357\mathbf{1270}\times10^{-3}$
Relative Difference	1.110×10^{-6}

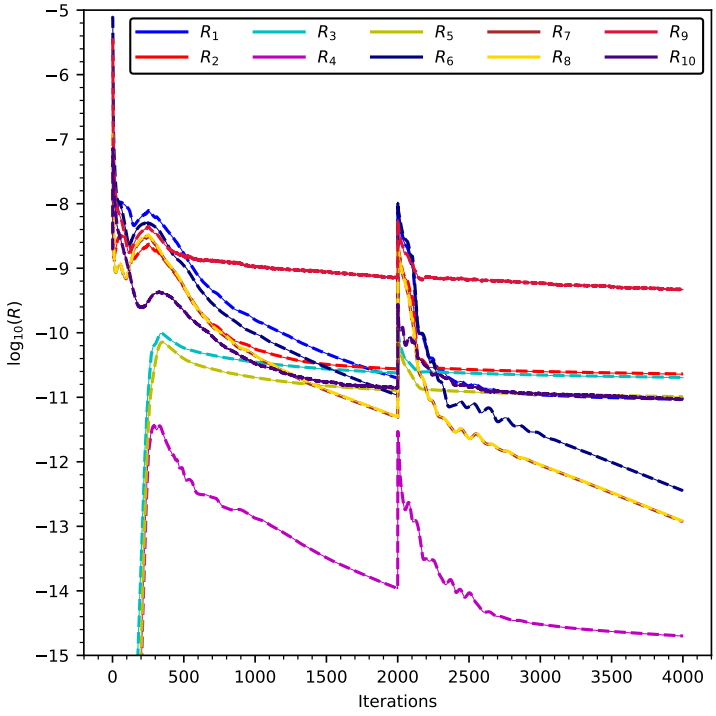


Fig. 14 Residual convergence history for hypersonic flow around a hemisphere cylinder, where CPU residuals are denoted by solid lines and GPU residuals are denoted by dashed lines.

Device-level performance comparisons for the hemisphere cylinder for various architectures are presented in Fig. 15. The architectures considered include Intel Xeon CPUs (Gold 6248 Cascade Lake and Gold 6148 Skylake), IBM POWER9 CPUs, AMD EPYC 7402 CPUs, and recent NVIDIA Tesla GPUs. Intel processor runs use the Intel Fortran 18.0.3 compiler with the flags "-Ofast -axCORE-AVX512 -ip -align -fno-alias" and the HPE MPT 2.23 MPI library. The POWER9 runs use the IBM XL Fortran for Linux 16.1.1 compiler with O3 optimizations and the IBM Spectrum 10.3.1 MPI library. The AMD EPYC runs use GCC 9.3.0 with the flags "-O3 -march=native -ffast-math" and the OpenMPI 4.0.3 MPI library. The plot is normalized to the NVIDIA Tesla V100 speedup; that is, a value of ten means a Tesla V100 is 10× faster than the indicated architecture.

As shown in the figure, an NVIDIA Tesla V100 outperforms a typical Intel Xeon Gold system by over an order of magnitude. The IBM POWER9 is nearly 15× slower than the Tesla V100, although it should be noted that FUN3D contains no specific optimizations for the POWER9 architecture, but is generally optimized for x86 architectures. An EPYC 7402 node is roughly 7× slower than the Tesla V100. An NVIDIA Tesla A100 outperforms the Tesla V100 by approximately 1.6× using code specifically tuned for the Tesla V100.

Like many large-scale science applications, FUN3D is memory bound. The memory bandwidth of the NVIDIA Tesla V100 is approximately 4.5× higher than a typical dual-socket Intel Xeon system. Therefore, the expected speedup is approximately 4.5× for well-optimized implementations on both architectures. While the performance of the perfect gas implementation is commensurate with this metric [11], it should be noted that the generic gas CPU implementation is not optimal. In particular, the computation of the generic gas flux Jacobians is extremely expensive on the CPU while the corresponding GPU implementation has been templated for each element type, the number of equations, and has been extensively optimized. The GPU speedup exceeds the memory bandwidth ratio for generic gas simulations for these reasons. It should also be noted that the NVIDIA Tesla A100 has approximately 70% higher memory bandwidth than the Tesla V100 and the corresponding performance ratio is close to this factor without tuning kernel launch parameters for the Tesla A100, which should further improve performance.

The results shown in Fig. 15 represent a nominal nonlinear iteration based on an evaluation of the residual vector and Jacobian matrix, the linear solve, and the solution update as given by Eqs. 31 and 32. This scenario strongly favors the GPU, as the GPU implementation of the flux Jacobians is significantly more efficient than that of the CPU. In practice, the Jacobian matrix is evaluated less frequently as the nonlinear system converges, whether in the context of a steady-state problem or within the subiterations of a single physical time step of an unsteady problem. With the flux Jacobians held constant, the linear solve instead dominates overall execution time. Since the speedup for the linear solve is much closer to the memory bandwidth ratio between the CPU and GPU, the overall GPU speedup for such time steps is less than that indicated in Fig. 15 (roughly 25% reduction in this case).

Other parameters can also affect the CPU/GPU performance ratio. For this case, using a one-temperature model results in a 10% relative reduction in performance for the GPU. Grid topology also affects performance. For typical grids used in FUN3D, the number of elements is approximately three to six times the number of nodes. Since the relatively expensive flux Jacobian kernels use element-based loops, these kernels and the linear solver consume similar proportions of the overall execution time. For grids with a similar number of elements and nodes (e.g., a purely hexahedral grid), this shifts the balance of the execution time toward the linear solver, which has a much lower GPU speedup. In addition, hexahedral grids have more neighbors per grid point than those consisting of other element types, which further increases the relative expense of the linear solve.

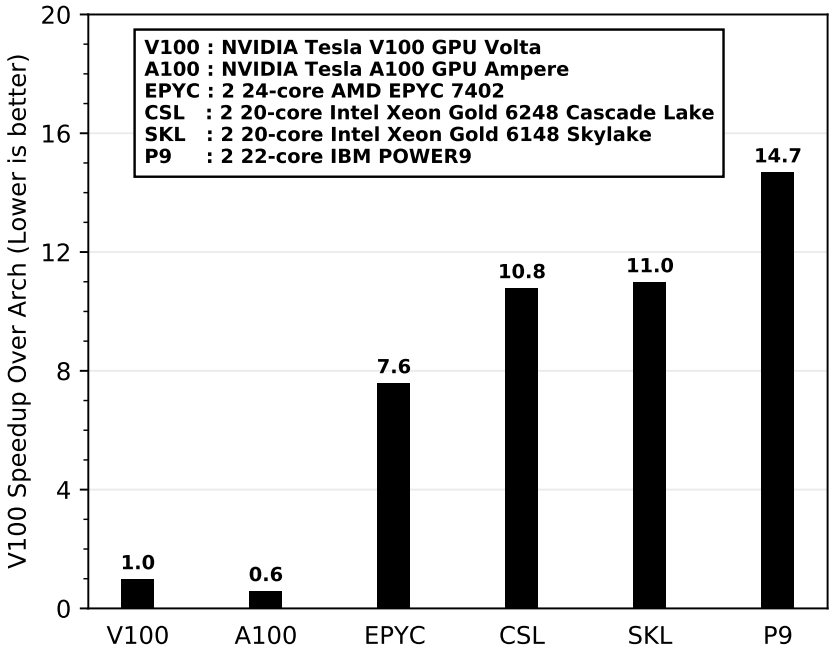


Fig. 15 Device-level performance comparison between different architectures for the hypersonic flow around a hemisphere cylinder (laminar viscous model, five species, two-temperature model). Lower is better. The value shown is the Tesla V100 speedup over the architecture. Thus, Tesla V100 is 11.0× faster than SKL for this case.

C. Transverse Reacting Hydrogen Jet in Supersonic Cross Flow

A transverse reacting hydrogen jet in supersonic cross flow (JISCF) is simulated. The conditions match experiments [35, 36] performed at the six inch Expansion Tube Facility of the High Temperature Gas Dynamics Laboratory at Stanford University. Standard air is injected at the inflow at Mach 2.4 with pure hydrogen injected from a round injector with diameter $D = 2$ mm. The injector geometry is taken from the experimental setup [35]. The jet-to-crossflow momentum flux ratio, $J = (\rho_j U_j^2)/(\rho_a U_a^2)$, is set to five, where a and j refer to crossflow and jet conditions, respectively. Experimental data primarily consist of OH planar laser-induced fluorescence (PLIF) imaging.

The unstructured grid is composed of 57,407,453 nodes and 299,957,560 cells (278,239,200 tetrahedra, 79,272 pyramids, and 21,639,088 prisms). The wall spacing is set to ensure a $y^+ \approx 1$ on the flat plate. The prismatic boundary layer grid is composed of 40 layers with a growth rate of 1.2. The remaining volume is meshed with tetrahedra. There is one volume source of $\Delta = 0.025D$ near the jet exit with another volume source of $\Delta = 0.05D$ sized approximately $8D \times 8D \times 40D$, where the larger dimension is extended toward the exit of the domain (40 diameters downstream). The extents of the domain are $72.5D \times 40D \times 20D$ in the axial, horizontal, and vertical directions, respectively. The case requires approximately 1.2 TB of memory at runtime.

A nine species (N_2,H_2,O_2,H,O,OH,H_2O,H_2O_2,HO_2), twenty reaction, hydrogen combustion mechanism is utilized [37] with a one-temperature model. The freestream Mach number is 2.4. The freestream temperature and pressure are $T_\infty = 1400$ K and $p_\infty = 40$ kPa, respectively. The freestream mass fractions are set to standard air. The Spalart-Allmaras turbulence model with the Catris-Aupoix compressibility correction [20] and the DES modification outlined in [1] is used. The wall is modeled as noncatalytic with a constant temperature of 300 K due to the short length of the tests (approximately 500 μs). The jet inlet is modeled as a pure hydrogen plenum with a total temperature and pressure of 300 K and 2.034 MPa, respectively, to match experimental conditions. Farfield is used on the inflow, side, and top planes. Extrapolation is used for the exit plane. The equations are integrated in time implicitly using BDF2 with 5 subiterations and a time step of $dt = 0.0063\tau_c$, where $\tau_c = D/u_\infty$. The global convective CFL number (the time step required for a purely explicit method neglecting chemistry, which imposes further restriction on stability) is approximately 25 based on the finest wall spacing and the speed of sound of hydrogen at the wall. The wake CFL number is around 1. A flow through time (taken as 40 τ_c) corresponds to approximately 6000 time steps. Statistics are collected for the duration of

160 τ_c after a statistically stationary state has been reached. Contours of various quantities are shown in Fig. 16. Jet features including upstream recirculation, bow shock, Mach disk, and mixing are successfully captured in the simulation. Jet height, computed based on peak mean OH mole fraction for the upper part of the flow field, versus experimental data is shown in Fig. 17. Results overall agree favorably to experimental OH-PLIF imaging data. The jet height is slightly overpredicted starting around $x/D = 6$. The recirculation region in front of the injector is also slightly smaller than the experiment, which may potentially be due to the turbulence model. Future work will investigate lower dissipation numerics as well as alternative turbulence models.

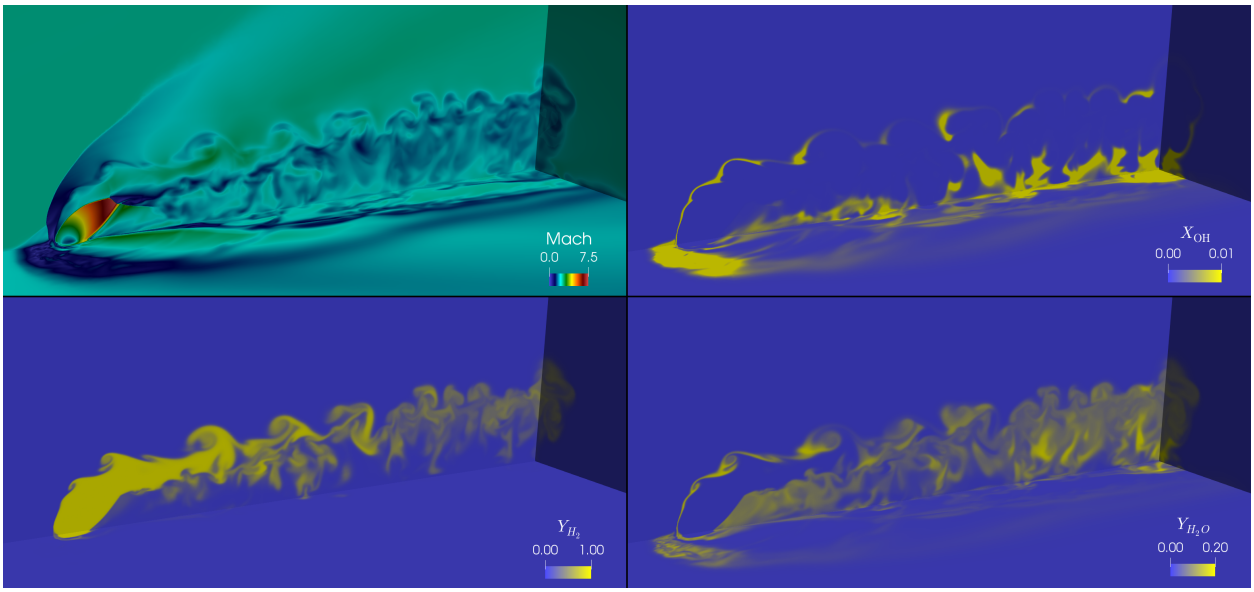


Fig. 16 Contours of the jet on the centerline, on a plane $z/D = 0.25$ above the plate, and on the exit plane. Top left: Mach number. Top right: OH mole fraction. Bottom left: H_2 mass fraction. Bottom right: H_2O mass fraction.

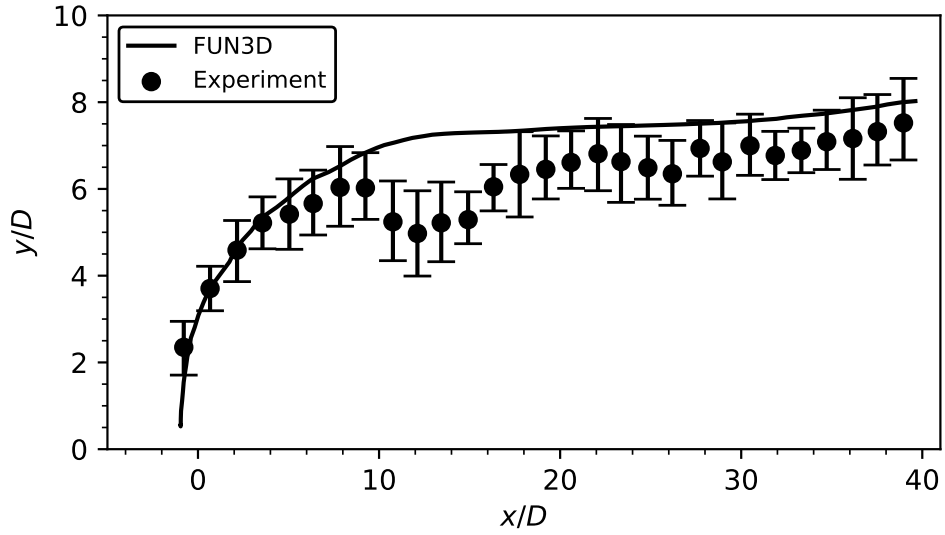


Fig. 17 Jet height versus axial distance. Jet height is taken as the peak in mean OH mole fraction for the upper part of the flow field. Experimental data with errors bars is from [36].

Table 4 shows the average wall-clock time taken by the GPU and CPU solvers to complete one time step and to simulate a flow through time. The GPU simulation uses eight nodes of six NVIDIA Tesla V100s (48 GPUs in total) whereas the CPU simulation uses 100 nodes of dual-socket 20-core Intel Xeon Gold 6148 Skylake CPUs (4,000 cores in total), using the same compiler setup as described for Fig. 15. The GPU result is obtained 4.6× faster than that of the CPU. It is estimated that the CPU system would require approximately 20,000 cores to match the time per iteration of the GPU system, implying that the efficiency of one Tesla V100 GPU is similar to that of 417 Skylake CPU cores. This is similar to the previous five species, two-temperature results for the hemisphere cylinder. The 1.2 TB of device memory required by the case uses nearly 80% of the 32 GB memory capacity of each GPU. The CPU system has 192 GB of memory for each node; less than 7% of the CPU memory is required to fit the simulation. This is a key limitation of GPUs for reacting flow simulations using the current implementation: the problem size, rather than the desired time to solution, dictates hardware requirements. Algorithmic improvements are currently being investigated to reduce the memory requirements of reacting flow simulations and to increase performance through memory-efficient iterations using approaches such as decoupled chemistry [38, 39] and turbulence (as is done for perfect gas flows in FUN3D).

Table 4 Reacting JISCF (nine species, one temperature, DES) CPU and GPU time comparisons for an iteration in seconds and flow through time (FTT) in hours.

	GPU (48 Tesla V100s)	CPU (4000 Skylake Cores)
Time/iteration [s]	5.6	25.8
Time/FTT [hrs]	9.3	43.0

D. Performance at Scale

The Summit system located at the Oak Ridge Leadership Computing Facility is an IBM architecture consisting of dual-socket 22-core POWER9 host processors and six NVIDIA Tesla V100 GPUs per compute node, as shown schematically in Fig. 18. With a total of 4,608 nodes, Summit debuted as the world's most powerful HPC platform in 2018 and remains the highest-ranked GPU system as of the time of this writing [40].

A series of representative grids for the Mars lander geometry shown in Fig. 1 is used to perform brief one-temperature reacting-gas simulations of unsteady turbulent flow with a 10 species, 19 reaction mechanism to capture weak scaling performance data on the Summit system. Here, a gas composition consisting of 97% CO_2 and 3% N_2 by mass is used to represent the Martian atmosphere and an equilibrium composition of the products of methane oxygen combustion is used for the eight nozzle plena with total temperature and pressure of approximately 3600 K and 80 bar, respectively. Grids are constructed such that approximately 1.2 million grid points are placed on each computational node. The computational resources and grid parameters for each simulation are shown in Table 5.

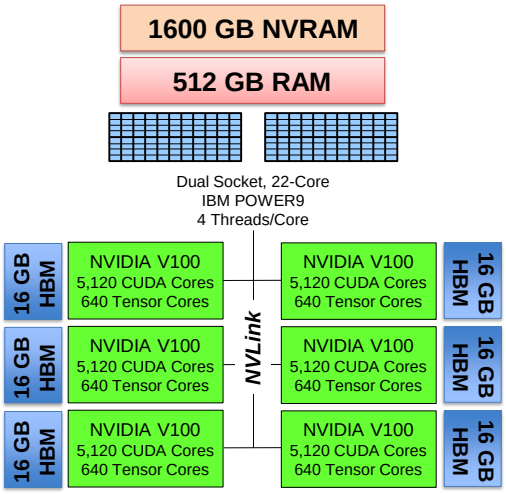


Fig. 18 Schematic representation of a Summit node.

Timings are established for a single physical time step consisting of 10 subiterations. Each simulation is first performed using only the CPUs available on each node, with a single MPI rank assigned to each processing core operating in SMT1 mode. The runs are then repeated using the six GPUs available per node, with a single MPI rank assigned to each. Fig. 19 plots the CPU and GPU scaling data versus the number of nodes used for each grid level. The GPU data show only small variations in scaling performance, notably absent at the highest node count, and a consistent speedup of 72-84x relative to the CPUs.

For the current problem, results indicate that a single NVIDIA Tesla V100 GPU is approximately 12.5× faster than a dual-socket IBM POWER9 processor at scale, similar to the results that were shown in Fig. 15. This factor suggests that the throughput achieved by the GPU-based simulation running on 1,018 Summit nodes is comparable to that of the CPU implementation running on approximately 3 million IBM POWER9 processor cores.

Table 5 Cases used for weak scaling analysis.

Summit Nodes	Total Number of GPUs	Grid Points	Prisms	Tetrahedra	Pyramids	Average Grid Points per Node
4	24	4,769,704	1,742,941	22,695,479	32,117	1,192,426
8	48	9,541,270	3,970,365	44,191,122	57,707	1,192,659
16	96	19,166,242	8,268,675	88,362,217	61,296	1,197,890
32	192	38,372,559	15,930,873	179,361,118	74,524	1,199,142
64	384	76,636,313	29,877,206	365,337,544	104,686	1,197,442
128	768	153,592,108	55,130,484	747,400,779	129,484	1,199,938
254	1,524	304,592,706	127,446,984	1,435,187,040	447,144	1,199,184
508	3,048	609,320,231	239,017,648	2,923,119,096	628,116	1,199,449
1,018	6,108	1,221,627,133	441,043,872	5,979,724,168	776,904	1,200,027

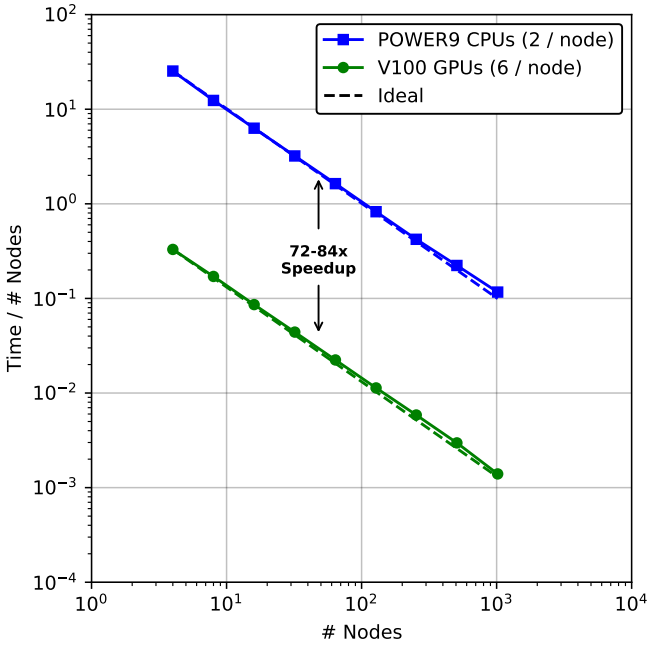


Fig. 19 CPU and GPU weak scaling performance on the Summit system.

V. Summary and Future Work

The FUN3D generic gas path has been successfully ported, optimized, and verified for NVIDIA Tesla GPUs. The governing equations for thermochemical nonequilibrium flows and the details of the numerical implementation have been described. A mini-app infrastructure was used for the primary GPU port with further development and additions incorporated in the full application. Conversion techniques and optimization methods have been discussed. The performance of the GPU-based generic gas implementation in FUN3D has been demonstrated and compared with the performance of the previously developed CPU-based generic gas implementation for several representative simulations. A representative hypersonic blunt body flow simulation at Mach 9.8 has been conducted on an unstructured grid using a two-temperature five species air model. For this example, one NVIDIA Tesla V100 GPU outperforms a dual-socket node comprised of two 20-core Intel Xeon Gold 6148 Skylake CPUs, reducing the computational time to solution by over an order of magnitude. A scale-resolving simulation of a chemically reacting JISCF using finite-rate kinetics has also been performed on GPUs. Such simulations have historically been prohibitively expensive to conduct on traditional CPU-based systems. For this simulation, one NVIDIA Tesla V100 GPU matches the performance of approximately 417 Intel Xeon Gold 6148 CPU cores at scale. The generic gas GPU implementation has also been benchmarked at extreme scale using as many as 6,108 GPUs on the Summit system with grids containing up to several billion elements. Performance at scale was observed to be equivalent to that of several million CPU cores.

Algorithmic improvements to the solver are considered for future work to improve accuracy and performance. These include, but are not limited to, lower dissipation numerics, decoupled approaches [38, 39], mixed-precision arithmetic [25], and nonlinear solver improvements [41, 42].

Acknowledgments

This research was sponsored by the NASA Langley Research Center CIF/IRAD program, NASA Transformational Tools and Technologies (TTT) Project of the Transformative Aeronautics Concepts Program under the Aeronautics Research Mission Directorate, and the National Institute of Aerospace Cooperative Agreement award NNL09AA00A. The authors would like to thank the Department of Defense High Performance Computing Modernization Program, NVIDIA, and Dr. Sameer Shende of the University of Oregon. 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. The authors would like to acknowledge contributions to this work from the following individuals at the October 2019 NVIDIA / OLCF GPU Hackathon: Dr. Li Wang (NASA Langley Research Center), Dr. Mohammad Zubair (Old Dominion University), Dr. Justin Luitjens (NVIDIA), and Mr. Bob Crovella (NVIDIA). The authors would also like to thank Dr. Robert Tramel (Kord Technologies), Dr. Boris Diskin (National Institute of Aerospace), and Dr. Kyle Thompson (NASA Langley Research Center) for helpful discussions pertaining to the current work.

References

[1] 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.

[2] Smagorinsky, J., "General Circulation Experiments with the Primitive Equations: I. The Basic Experiment," *Monthly Weather Review*, Vol. 91, No. 3, 1963, pp. 99–164.

[3] 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 November 20, 2020.

[4] 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 November 20, 2020.

[5] Howard, M., Fisher, T. C., Hoemmen, M. F., Dinzl, D. J., Overfelt, J. R., Bradley, A. M., and Kim, K., "Employing Multiple Levels of Parallelism for CFD at Large Scales on Next Generation High-Performance Computing Platforms." Tech. rep., Sandia National Laboratories, 2018.

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

[7] ORNL, "Frontier," https://www.olcf.ornl.gov/frontier, 2020. Last Accessed November 20, 2020.

[8] Thomas, J., "El Capitan," https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer, 2020. Last Accessed November 20, 2020.

[9] 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.

[10] 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: 13.6," *NASA TM-2019-220416*, 2019.

[11] 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.

[12] Korzun, A. M., Cruz, J. R., and Braun, R. D., "A Survey of Supersonic Retropropulsion Technology for Mars Entry, Descent, and Landing," *2008 IEEE Aerospace Conference*, IEEE, 2008, pp. 1–15.

[13] Korzun, A. M., Nielsen, E. J., Walden, A. C., Jones, W. T., Carlson, J.-R., Moran, P. J., Henze, C. E., and Sandstrom, T. A., "Effects of Spatial Resolution on Retropropulsion Aerodynamics in an Atmospheric Environment," *AIAA SciTech 2020 Forum 2020-1749*, 2020.

[14] Korzun, A. M., Nielsen, E., Walden, A., Jones, W., Carlson, J.-R., Moran, P., Henze, C., and Sandstrom, T., "Computational Investigation of Retropropulsion Operating Environments with a Massively Parallel Detached Eddy Simulation Approach," *ASCEND AIAA 2020-4228*, 2020.

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

[16] Boussinesq, J., *Essai sur la théorie des eaux courantes*, Impr. nationale, 1877.

[17] 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.

[18] 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.

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

[20] Catris, S., and Aupoix, B., "Density Corrections for Turbulence models," *Aerospace Science and Technology*, Vol. 4, No. 1, 2000, pp. 1–11.

[21] Roe, P. L., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, No. 2, 1981, pp. 357–372.

[22] Burg, C., "Higher Order Variable Extrapolation For Unstructured Finite Volume RANS Flow Solvers," *17th AIAA Computational Fluid Dynamics Conference*, AIAA Paper 2005–4999.

[23] Van Leer, B., "Towards the Ultimate Conservative Difference Scheme. V. A Second-Order Sequel to Godunov's Method," *Journal of Computational Physics*, Vol. 32, No. 1, 1979, pp. 101–136.

[24] 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 (IA3)*, IEEE, 2016, pp. 18–25.

[25] Walden, A., Nielsen, E., Diskin, B., and Zubair, M., "A Mixed Precision Multicolor Point-Implicit Solver for Unstructured Grids on GPUs," *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, IEEE, 2019, pp. 23–30.

[26] The MPI Forum, "The MPI Forum Website," http://www.mpi-forum.org, 2020. Last Accessed November 20, 2020.

[27] The OpenMP Consortium, "The OpenMP API," https://www.openmp.org, 2020. Last Accessed November 20, 2020.

[28] NVIDIA Corporation, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide, 2020. Last Accessed November 20, 2020.

[29] NVIDIA, "Multi-Process Service," https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2020. Last Accessed November 20, 2020.

[30] Heroux, M., Doerfler, D., Crozier, P., Willenbring, J., Edwards, H., Williams, A., Rajan, M., Keiter, E., Thornquist, H., and Numrich, R., "Improving Performance via Mini-Applications," *Sandia National Laboratories Technical Report SAND2009-5574*, 2009.

[31] Karl, S., Martinez Schramm, J., and Hannemann, K., "High Enthalpy Cylinder Flow in HEG: A Basis for CFD Validation," *33rd AIAA Fluid Dynamics Conference and Exhibit 2003-4252*, 2003.

[32] Park, C., "On Convergence of Computation of Chemically Reacting Flows," *23rd Aerospace Sciences Meeting 1985-247*, 1985.

[33] Knight, D., Longo, J., Drikakis, D., Gaitonde, D., Lani, A., Nompelis, I., Reimann, B., and Walpot, L., "Assessment of CFD Capability for Prediction of Hypersonic Shock Interactions," *Progress in Aerospace Sciences*, Vol. 48, 2012, pp. 8–26.

[34] Josyula, E., and Shang, J. S., "Computation of Nonequilibrium Hypersonic Flowfields around Hemisphere Cylinders," *Journal of Thermophysics and Heat Transfer*, Vol. 7, No. 4, 1993, pp. 668–679.

[35] Gamba, M., Terrapon, V., Saghafian, A., Mungal, M. G., and Pitsch, H., "Assessment of the Combustion Characteristics of Hydrogen Transverse Jets in Supersonic Crossflow," *Annual Research Briefs*, 2011, pp. 259–272.

[36] Gamba, M., and Mungal, M. G., "Ignition, Flame Structure and Near-Wall Burning in Transverse Hydrogen Jets in Supersonic Crossflow," *Journal of Fluid Mechanics*, Vol. 780, 2015, pp. 226–273.

[37] Jachimowski, C. J., "An Analytical Study of the Hydrogen-Air Reaction Mechanism with Application to Scramjet Combustion," *NASA TP 2791*, 1988.

[38] Candler, G. V., Subbareddy, P. K., and Nompelis, I., "Decoupled Implicit Method for Aerothermodynamics and Reacting Flows," *AIAA Journal*, Vol. 51, No. 5, 2013, pp. 1245–1254.

[39] Thompson, K. B., and Gnoffo, P. A., "A Decoupled Method for the Roe FDS Scheme in the Reacting Gas Path of FUN3D," *54th AIAA Aerospace Sciences Meeting 2016-2062*, 2016.

[40] Strohmaier, E., Dongarra, J., Simon, H., Meuer, M., and Meuer, H., "The Top 500 List," http://www.top500.org, 2020. Last Accessed November 20, 2020.

[41] Pandya, M. J., Diskin, B., Thomas, J. L., and Frink, N. T., "Assessment of USM3D Hierarchical Adaptive Nonlinear Method Preconditioners for Three-Dimensional Cases," *AIAA Journal*, Vol. 55, No. 10, 2017, pp. 3409–3424.

[42] Wang, L., Diskin, B., Nielsen, E., and Liu, Y., "Improvements in Iterative Convergence of FUN3D Solutions," *AIAA SciTech Forum*, 2021.