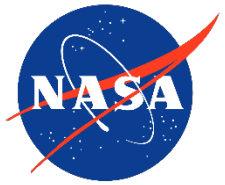


A Multi-Architecture Approach for Implicit Computational Fluid Dynamics on Unstructured Grids

Gabriel Nastac, Aaron Walden, Eric Nielsen
and FUN3D Team

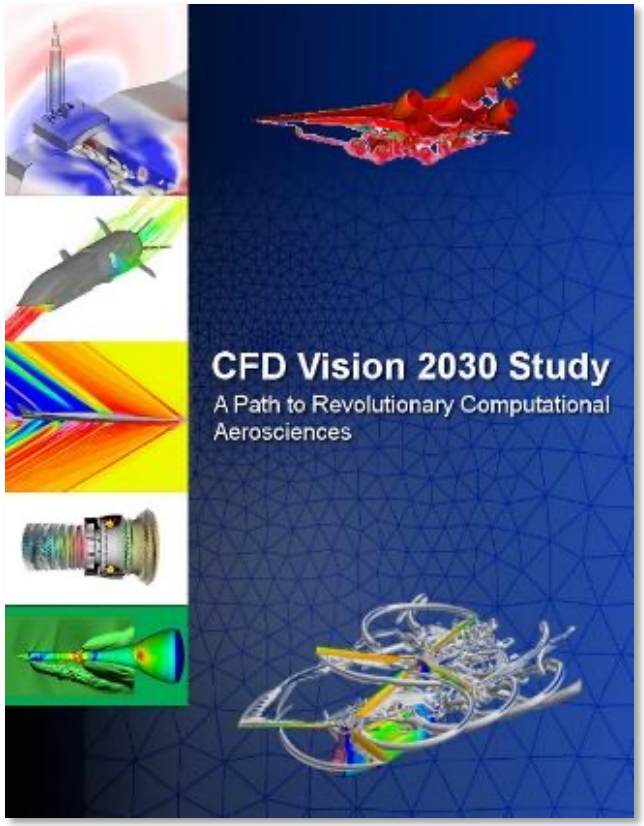
NASA Langley Research Center

Overview

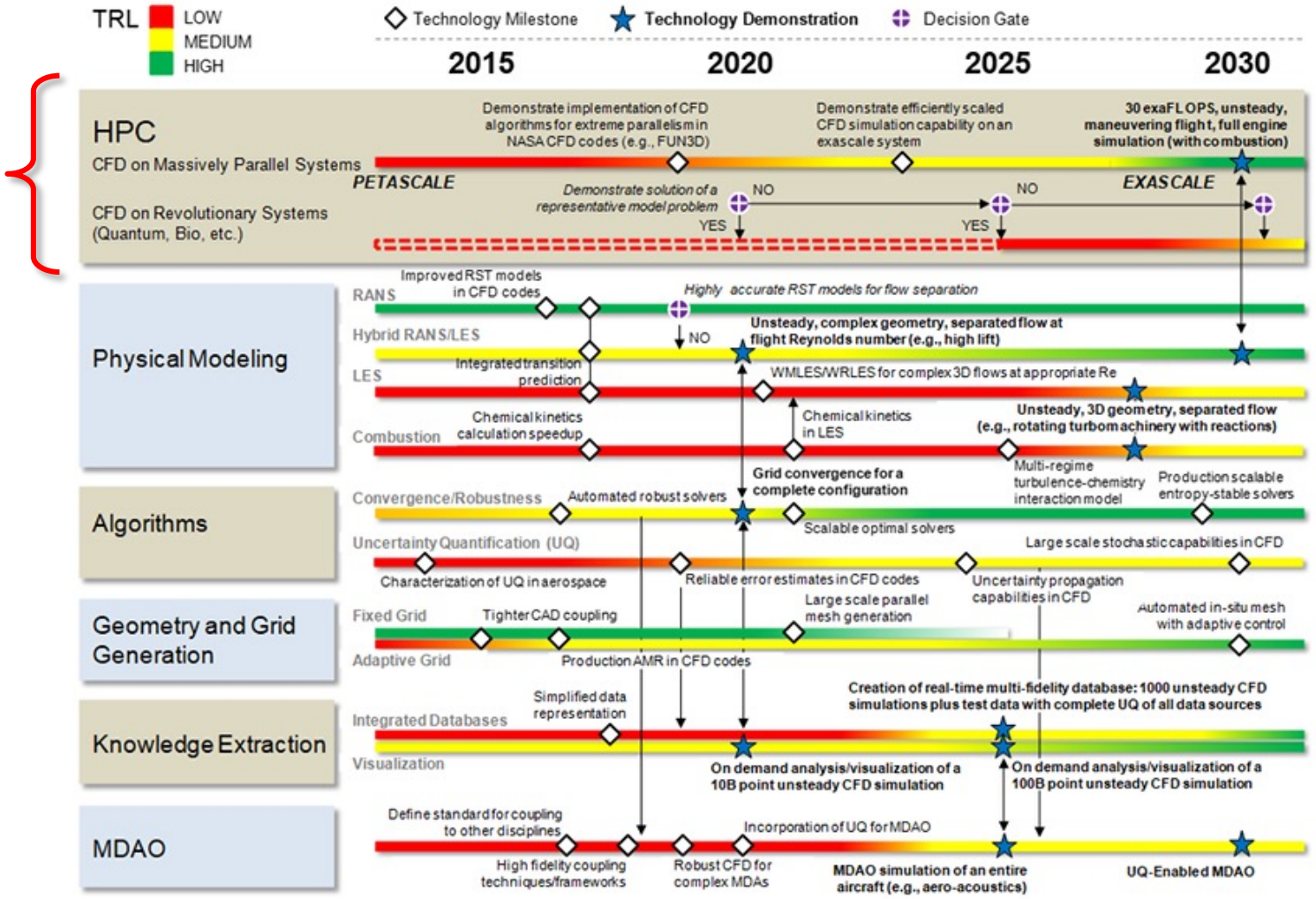
- Motivation
  - Why Graphics Processing Units (GPUs)?
- Overview of FUN3D CFD Solver and Applications
- Overview of GPUs and GPU Programming
- Multi-Architecture FUN3D and Simulations (SciTech 2023 work[1])
- Supersonic Retropropulsion Trajectory Simulations (Aviation 2023 work[2])
- Summary and Future Work

[1]https://arc.aiaa.org/doi/abs/10.2514/6.2023-1226
[2]https://arc.aiaa.org/doi/abs/10.2514/6.2023-3693

The CFD Vision 2030 Study



NASA/CR-2014-218178
See https://www.cfd2030.com



A Wake-Up Call for US Supercomputing

https://commons.wikimedia.org/wiki/File:EarthSimulator.jpg; Manatee_tw; Earth Simulator in Japan JAMSTEC 2007; https://creativecommons.org/licenses/by-sa/2.0/

New York Times Front Page, April 20, 2002:

"Japanese Computer is World's Fastest, As U.S. Falls Back"

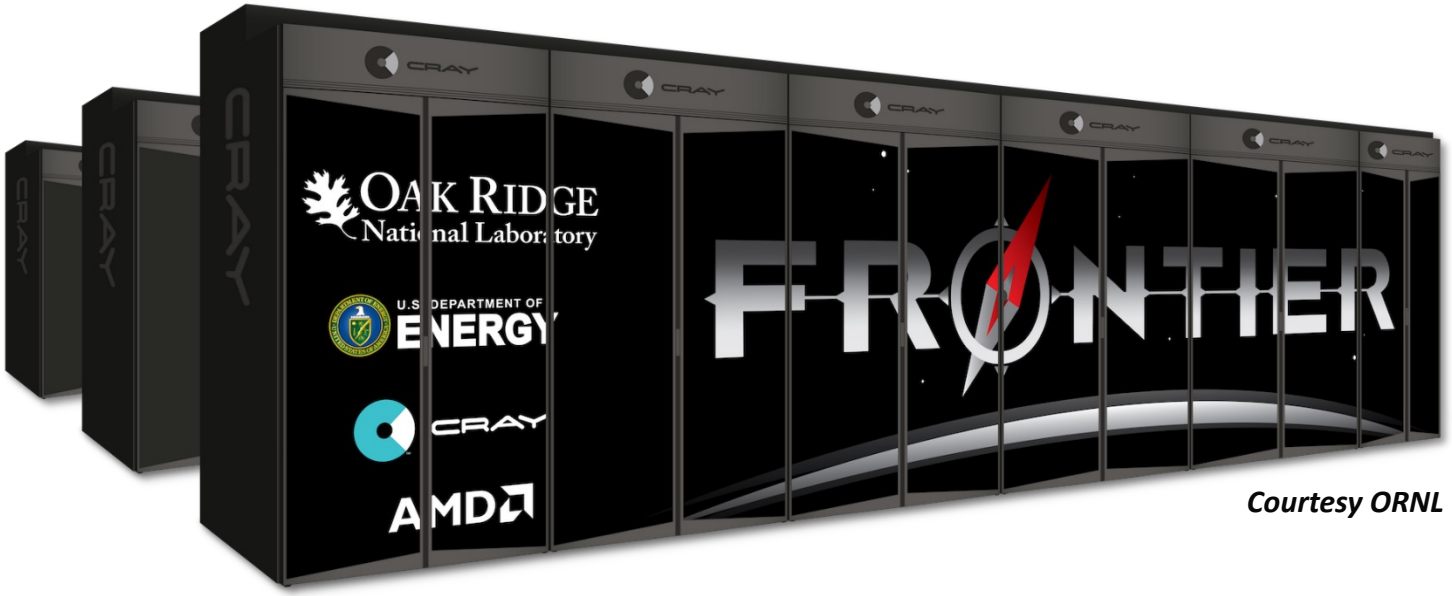
"A Japanese laboratory has built the world's fastest computer, a machine so powerful that it matches the raw processing power of the 20 fastest American computers combined…"

"For some American computer scientists, the arrival of the Japanese supercomputer evokes the type of alarm raised by the Soviet Union's Sputnik satellite in 1957."

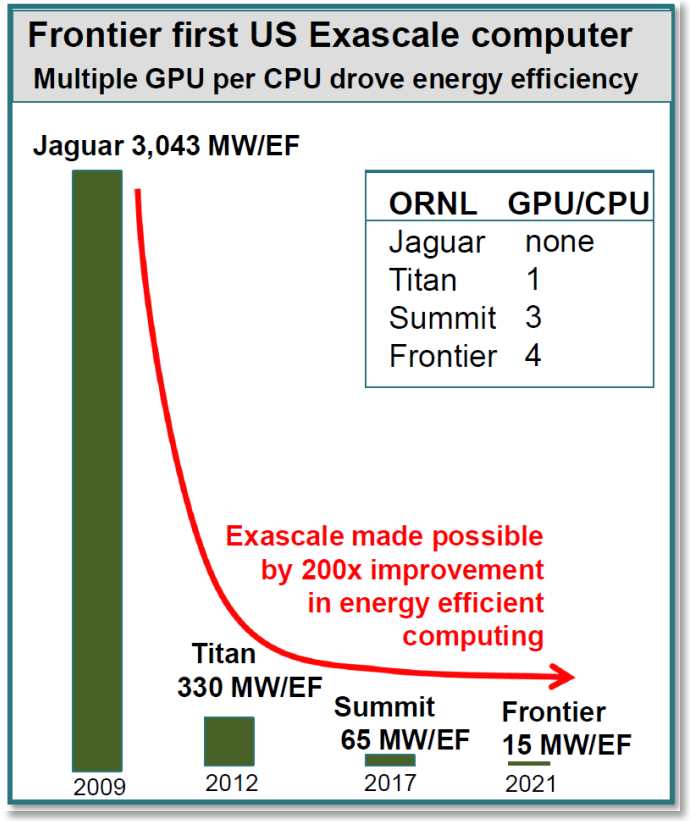
Computational speed: 36 Teraflops, or 36 x 10^{12} operations per second

Summer 2022: The Exascale Era Has Arrived

With the latest GPUs, we can now hold the 2002 Japanese system in the palm of our hand



Courtesy ORNL



Courtesy Justin Whitt, ORNL

Using 38,000 such processors, the ORNL Frontier system officially broke the exascale barrier this last summer:
1.1 exaflops, or 10^{18} operations per second

3000 MW ~ 200,000 Electric Cars on HW

15 MW ~ 1,000 Electric Cars on HW

Motivation

- GPUs are the key technology for next-generation HPC
  - **Seven of the top 10 supercomputers (based on TOP500) use GPUs**
  - **The top 10 supercomputers represent half of the computing power of the TOP500**
  - **Half of the top 100 supercomputers in the world use GPUs**
  - **Top 10 Green500 (power efficiency list) systems all use GPUs**
- U.S. exascale systems rely on GPU acceleration
- GPUs enable faster and more efficient simulations at all fidelities
  - Most design and analysis today employs Euler and RANS simulations
  - GPUs can better enable scale-resolved simulation use in design and analysis
- Graphics processing units (GPUs) have a few orders of magnitude greater concurrency than multicore CPUs
  - NVIDIA A100 GPU: 221,184 logical threads
  - AMD EPYC 7742 CPU: 256 logical threads
- Extracting potential hardware performance requires exposing more parallelism

Current HPC Landscape (June 2023)
Org Name (Rmax) Installation Year
1. ORNL Frontier (1200 PF) 2022
5. ORNL Summit (150 PF) 2018
100. DoD Onyx (6 PF) 2017

Upcoming US Systems in 2023
ANL Aurora (2000 PF)
LLNL El Capitan (2000 PF)

Architecture: CPU / GPU

PF: PetaFLOPS, or 10^{15} Floating-Point Operations Per Second

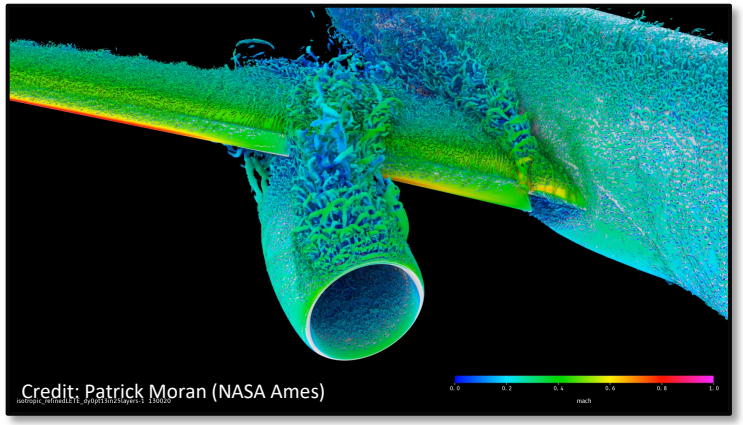
Euler Equations
(inviscid)

Reynolds-averaged Navier-Stokes: "RANS"
(turbulence modeled)

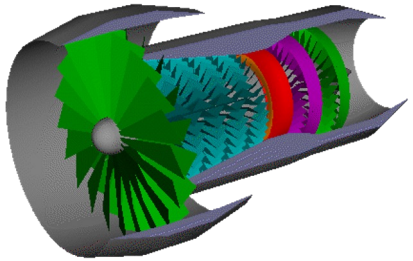
Large-Eddy Simulations: "LES"
(large scales resolved)

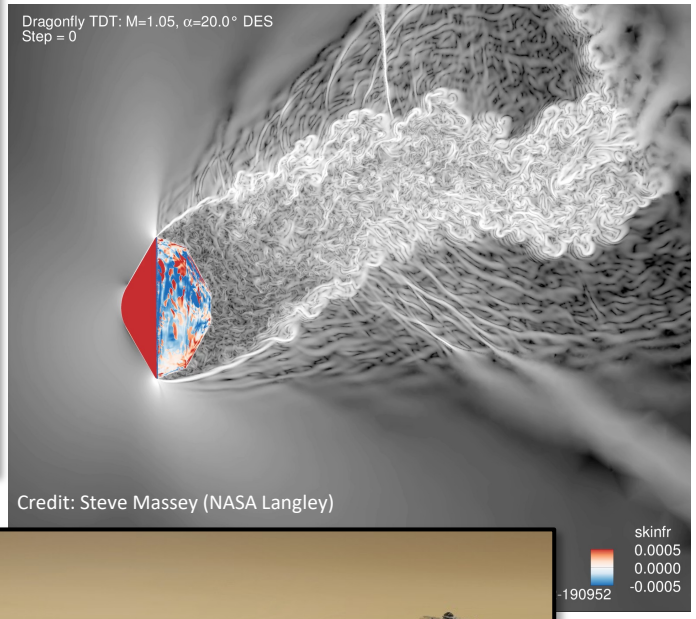
Direct Numerical Simulations: "DNS"
(all scales resolved)

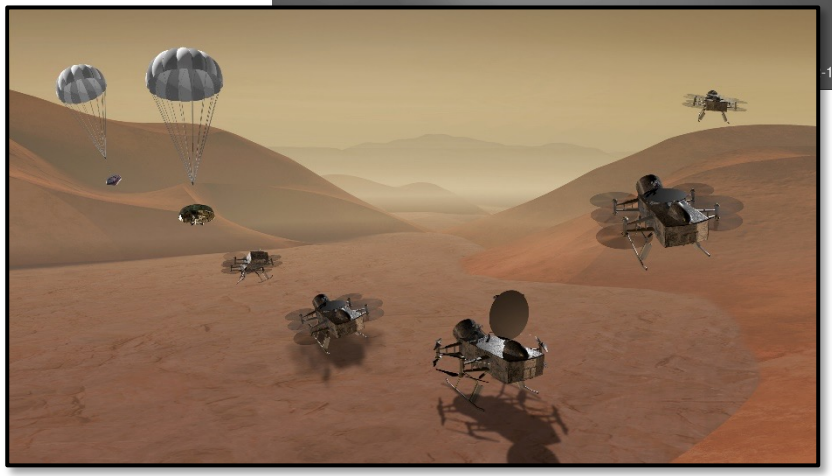
Increasing physics, increasing cost ⟶



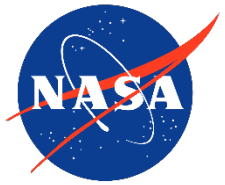
Emerging Architectures: Why are they important?







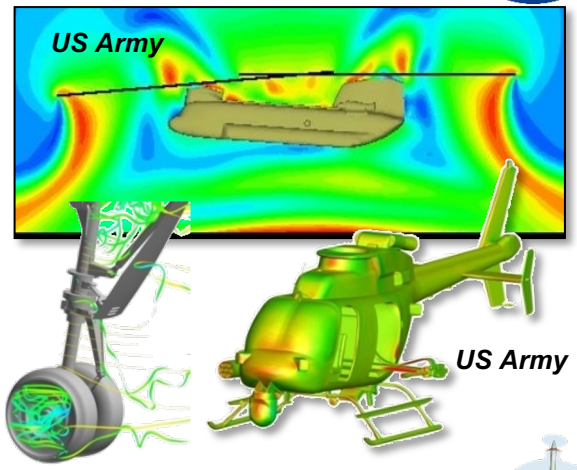
- Certification by Analysis: Substantial savings for development programs
- Rapid assessment and solutions for anomalies seen in testing
- Novel and more efficient vehicle designs
- High-fidelity sims where conventional methods are inaccurate or where testing is infeasible
- UQ, MDAO, robust design, maneuvers and trajectories…

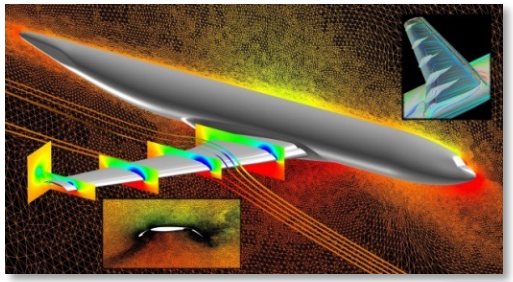


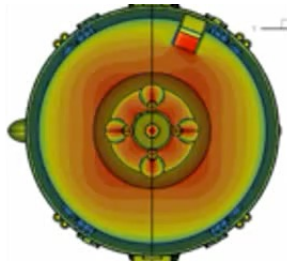
Overview of FUN3D CFD Solver and Applications

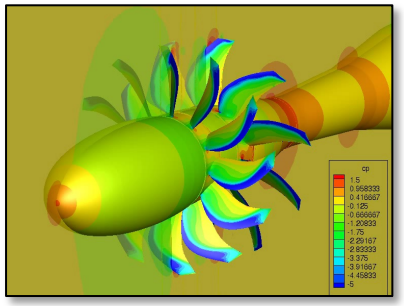
NASA Fully Unstructured Navier-Stokes 3D (FUN3D)

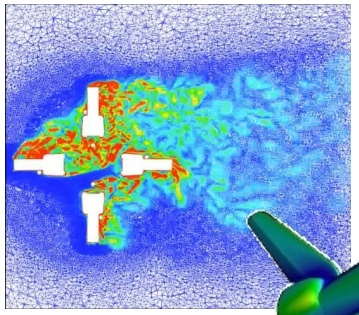
- Established as a research code in late 1980s; now supports numerous internal and external efforts across the speed range
- **Solves 3D Navier-Stokes equations using node-based finite-volume approach on mixed element unstructured grids**
- Fully **implicit formulations** are generally used to integrate the equations
- General dynamic mesh capability: any combination of rigid / overset / morphing grids, including 6-DOF effects
- Aeroelastic modeling using mode shapes, full FEM, CC, etc.
- Constrained / multipoint adjoint-based design, mesh adaptation
- Distributed development team using agile/extreme software practices including 24/7 regression, performance testing
- Capabilities fully integrated, online documentation, training videos, tutorials
- **Recent multi-architecture capabilities enables running performantly on NVIDIA/AMD/Intel GPUs and multicore CPUs using a primarily single-source codebase**
- https://fun3d.larc.nasa.gov

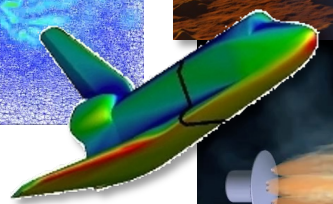


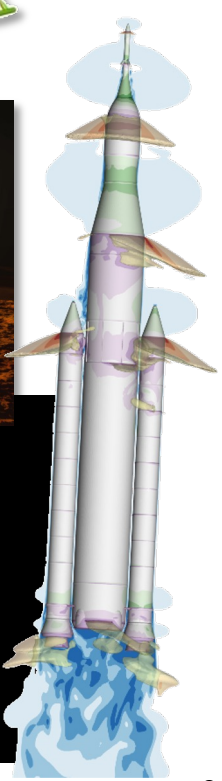


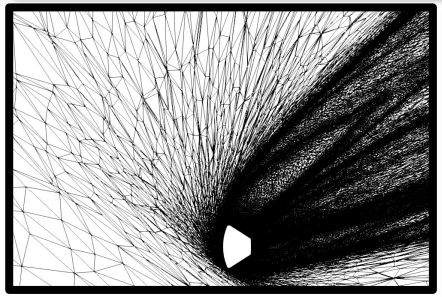


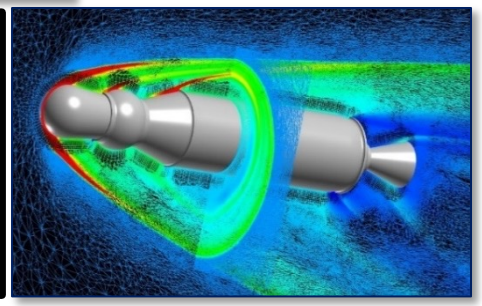


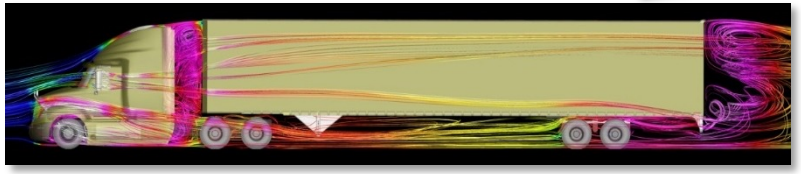


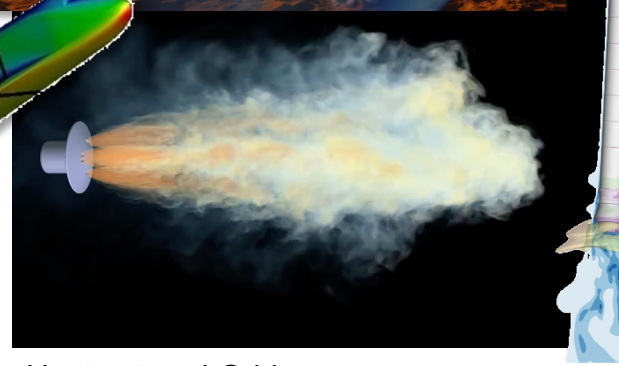






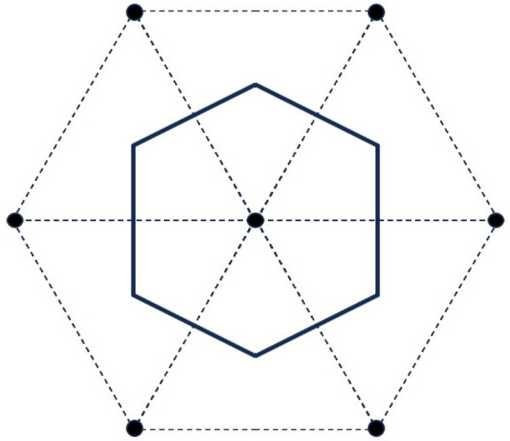






Governing Equations and Numerical Implementation

- Conservation of species, momentum, energies, and turbulence variables
- Variable species, energies, and turbulence equations
- Node-based finite-volume approach on general unstructured grids
- Fully implicit formulations are used to integrate the equations in time
  - Sparse block linear system: $A\boldsymbol{x} = \boldsymbol{b}$
  - Matrix A composed of diagonal and off-diagonal $N_{eq}\,x\,N_{eq}$ Jacobian blocks
    - Jacobian = derivative of vector function (N_{eq})
  - Memory and solution time increases as $O\left(N_{eq}^2\right)$
- System typically solved with multicolor point-implicit approach



Node-based finite volume in 2D

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

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

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

$$\frac{\partial}{\partial t}(\rho E_v) + \frac{\partial}{\partial x_j}(\rho E_v u_j) - \frac{\partial}{\partial x_j}\left(\dot{q}_{V\,j} + \sum_{s=1}^{N_s} h_{V\,s} J_{sj}\right) = S_v$$

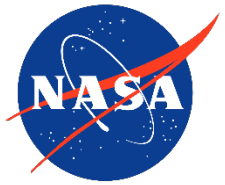
$$\frac{\partial}{\partial t}(\rho\tilde{\nu}) + \frac{\partial}{\partial x_j}(\rho\tilde{\nu}u_j) - \frac{\partial}{\partial x_j}\left(\frac{1}{\sigma}\left(\mu\frac{\partial\tilde{\nu}}{\partial x_j} + \sqrt{\rho}\tilde{\nu}\frac{\partial\sqrt{\rho}\tilde{\nu}}{\partial x_j}\right)\right) = S_{\tilde{\nu}}$$

$$\boldsymbol{q} = [\rho\vec{y}_s, \rho\vec{u}, \rho E, \rho E_v, \rho\tilde{\nu}]^T$$

$$\int_V \frac{\partial \boldsymbol{q}}{\partial t} dV + \oint_S (\boldsymbol{F}\cdot\boldsymbol{n}) dS - \int_V \boldsymbol{S} dV = \boldsymbol{0}$$

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

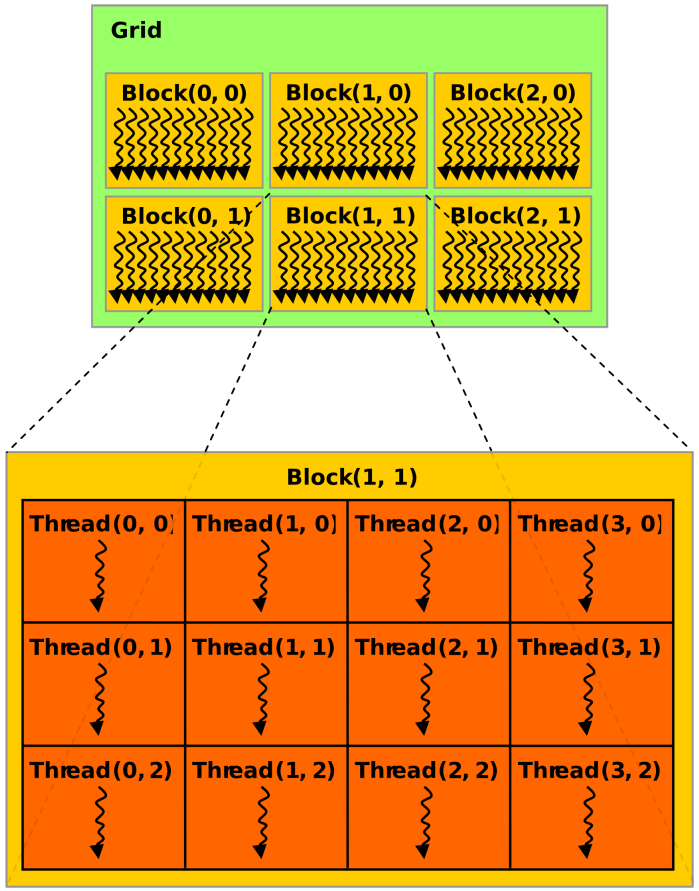
$$\mathbf{q}^{n+1,m} = \mathbf{q}^{n+1,m} + \Delta\boldsymbol{q}$$



Overview of GPUs and GPU Programming

Overview of GPUs [1/2]

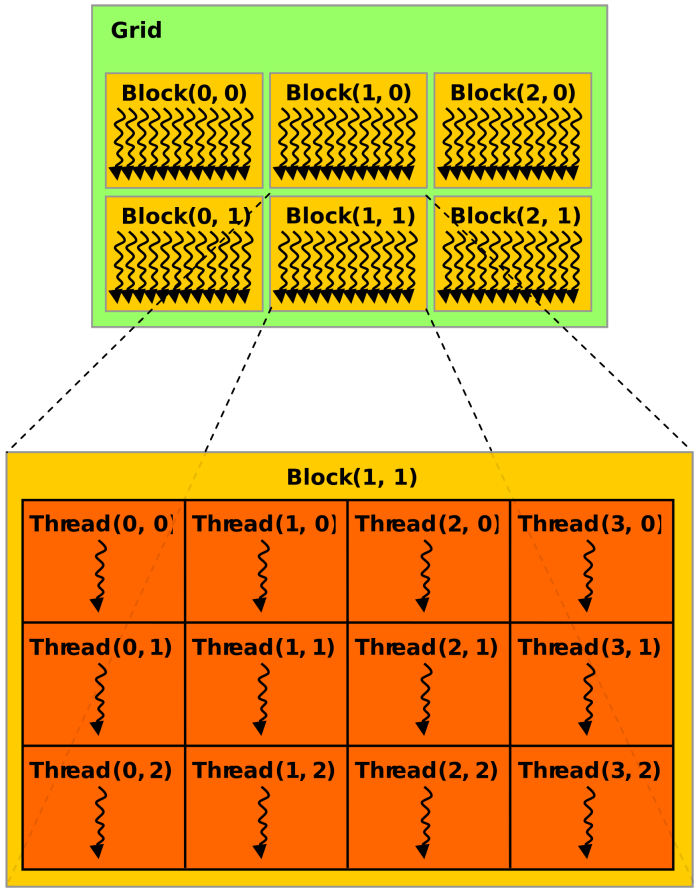
- Terminology is NVIDIA-specific, but NVIDIA, AMD, and Intel GPUs are all fundamentally similar
- Single-Instruction Multiple-Thread (SIMT) Paradigm
- Groups of threads (warp of 32 threads for NVIDIA) compute at the same time in lock-step
  - Ideal for problems where you want to compute the same thing for many items with little divergence
- GPUs are throughput machines; have higher latency than CPUs which is overcome with many threads
  - GPU ~ slow train
  - CPU ~ fast car
- NVIDIA A100 GPU:
  - 221,184 logical threads
- Kernels (functions) operate on a grid of threads
- Grid composed of Blocks composed of Threads



https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

Overview of GPUs [2/2]

- Memory Hierarchy
  - Thread-local memory (“registers”)
  - Block/shared memory
  - Global memory
- Local/shared memory O(100) times faster than global
- **Latency is hidden through oversubscription of threads**
  - **processors can switch between warps in one clock cycle**
  - **process warps with data ready**
- O(100) doubles of memory per thread for high-end GPUs
- Limited block/shared memory as well
- Exceeding these sizes leads to ”spilling” and use of global memory which is very slow
- 100 variables is not a lot, especially for implicit problems
  - 5 species air = 9 equations → 81 variables for a Jacobian, ignoring intermediate data
- Solution: expose more parallelism. Hierarchical parallelism
- More threads per item increase our available fast memory and also better hides latency



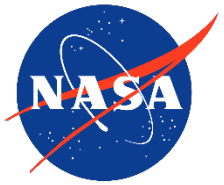
https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

Memory and Compute Bound Problems

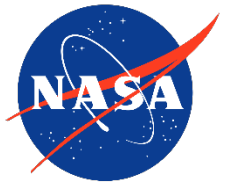
- Loading data from main memory takes a long time
- Arithmetic Intensity (AI) = work / memory traffic (FLOPs/byte)
  - Low AI → Limited by Memory Bandwidth (MBW)
  - High AI → Limited by FLOPs
- Many CFD algorithms including 2nd order FVM have low AI
- **GPUs are conveniently very efficient for memory bound problems**
- Strictly looking at reported hardware specifications to get a rough estimate
  - Disclaimer: specifications are not necessarily obtained in practice and vary between architectures
  - Ignores hosts for GPUs, but typically GPU nodes have at least 4-8 GPUs/node if not more
  - **A single 8x high-end GPU node (size of a big desktop tower) is equivalent to thousands of current CPU cores of performance**

Architecture: CPU / GPU	MBW GB/s	FP64 TF	Watt	Ratio MBW/W	Ratio TF/W
Intel Skylake 6148 Dual-Socket (40-core)	256	3	300	1	1
AMD EPYC 7762 Dual-Socket (128-core)	409.6	7	450	1.1	1.6
Intel Sapphire Rapids 9480 HBM Dual-Socket (112-core)	2000	8.2	700	3.3	1.2
AMD EPYC 9654 Dual-Socket (192-core)	921.6	10.8	720	1.5	1.5
NVIDIA V100	900	4.4	300	3.5	1.5
NVIDIA A100	2039	9.7	300	7.8	3.3
NVIDIA H100-NVL	7800	68	700	13.1	9.7
AMD MI250X	3276.8	48	560	6.9	8.6
AMD MI300X	5218	TBD	850	7.2	TBD
Intel Data Center GPU Max 1550	3276.8	52	600	6.4	8.7

Programming on GPUs

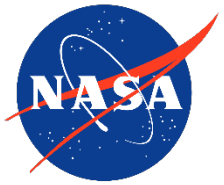


- Difficult or impossible to compile and run legacy software on GPUs efficiently without significant refactoring
- Most vendors have vendor-specific paradigms
- Many programming models exist to run on GPUs, e.g., (not exhaustive)
  - NVIDIA CUDA C++, AMD HIP, Intel DPC++, OpenCL, ISO C++, Vulkan Compute, SYCL, OpenMP, OpenACC, Kokkos, RAJA
- **Fundamentally, many models utilize a SIMT-like paradigm and are very similar**
  - **Many models have converter scripts from other models**
- **Models do not automatically parallelize your code efficiently, onus is still on the developer**
- Ideally, one would write in a standardized specification supported by major hardware vendors and achieve satisfactory performance across contemporary HPC architectures
- **Performance is paramount for GPU adoption; it must be cost-effective and performant enough to potentially rearchitect software**
  - **Especially so as cloud computing becomes more prevalent**
  - **If it is cheaper to run on CPUs (which it shouldn't be for optimal implementations), people will do so**



Multi-Architecture FUN3D and Simulations

FUN3D Library for Universal Device Acceleration (FLUDA)

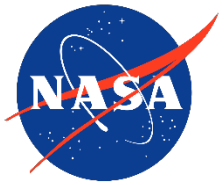


- **Finite-volume on unstructured grids is primarily memory-bound, so performance should scale with memory bandwidth to 1st order**
  - Limited success with early GPU efforts to achieve performance parity with highly optimized legacy Fortran
- To achieve high performance on NVIDIA GPUs, FLUDA was created in 2017 as a CUDA port of FUN3D's flow solvers
  - Identical data structures to Fortran
  - Double precision variables
- AMD GPU support was extended using a thin abstraction layer similar to CUDA C++ and AMD HIP
- Abstraction enables:
  - NVIDIA GPU usage through CUDA C++
  - AMD GPU usage through AMD HIP
  - Intel GPU usage through SYCL
  - CPU usage through ISO C++
- CPUs run the GPU-oriented code with a single thread
- Preprocessing macros comprise ~500 lines of code
- **Our hypothesis is that modern, superscalar CPUs will better tolerate GPU-oriented code than the reverse**

```
#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);
   }
}
```

Example FLUDA edge (dual-face) kernel

GPU Design Approach



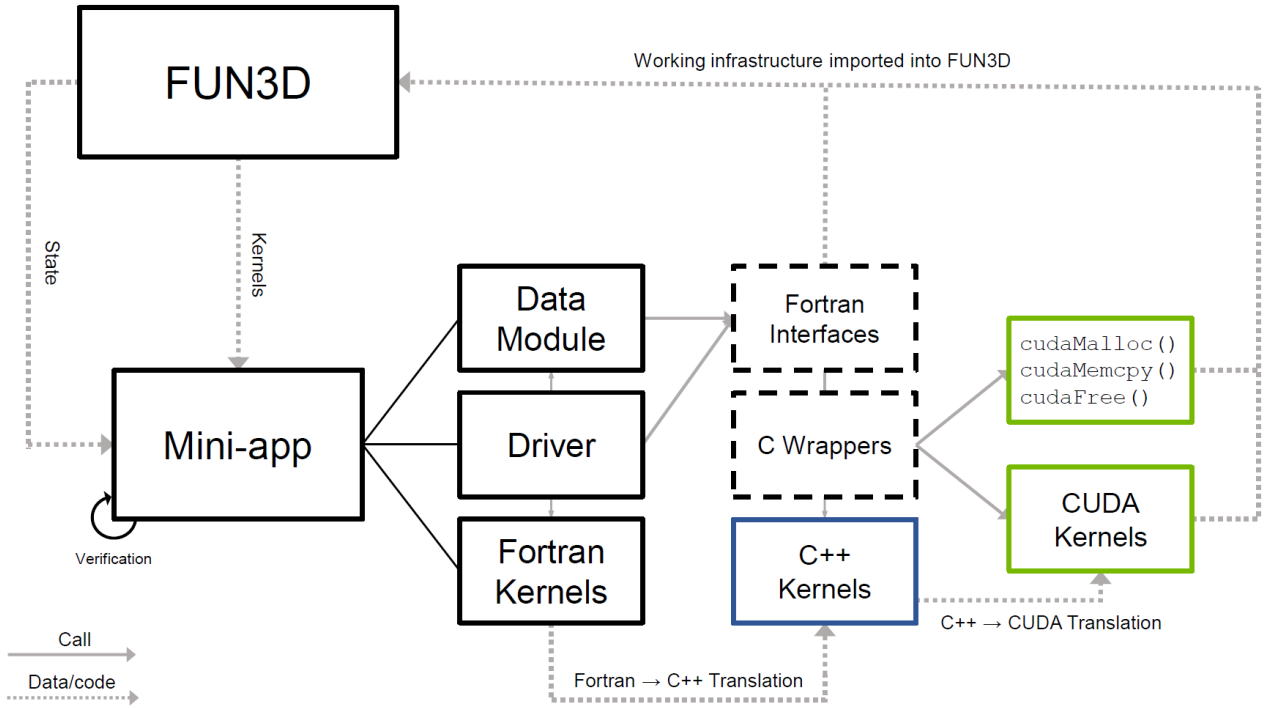
- Nomenclature: Host = CPU, Device = GPU
- Originally CUDA C++ port of compute kernels in FUN3D, now use thin abstractions above CUDA for all GPUs/CPUs
- No external libraries are required
- Effectively C with templates, often reads like Fortran
  - Simplest, most straight forward code possible helps compilers
  - ”Data-oriented Design”
- Compile the library N times for N architectures; load dynamically at run-time for chosen architecture
- Use of library in FUN3D is controlled by a run-time parameter
  - At the top of a Fortran FUN3D kernel, we call the FLUDA kernel and return, otherwise Fortran kernel runs
  - This does mean we currently have two implementations (Fortran and multi-architecture C++) which poses many challenges although enables continual support and usage for projects that rely on the solver
  - Moving forward, multi-architecture C++ implementation will be used for CPU runs too and Fortran will be deprecated
  - Most primary/common solver paths and options have been adapted at this time
    - incompressible gas, perfect gas, generic gas, moving grids, aeroelasticity, RANS/DES turbulence models, WMLES

GPU Design Approach (cont.)

- Pre-processing and post-processing routines remain on the host
  - Slow when GPUs are O(100-1000s) cores of performance
  - Often done just once per run
- All PDE kernels performed on device
- Minimal data transfer between host/device (mainly scalars)
  - Large data motion at user-specified frequencies (e.g., restarts, visualization support)
- Data structures are identical between FLUDA and Fortran contexts
  - Column-major order array layouts
  - Mostly arrays
  - GPU “mirror” data structures that match CPU data structures
  - Variable precision is identical to CPU approach
  - **Discrete consistency to double precision between Fortran and FLUDA across architectures**

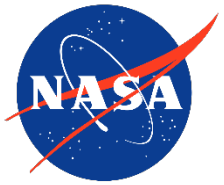
Architecture	Implementation	C_D
Intel Skylake 6148	Fortran	1.435840192×10^{-2}
Intel Skylake 6148	FLUDA	1.435840192×10^{-2}
AMD EPYC 7742	Fortran	1.435840192×10^{-2}
AMD EPYC 7742	FLUDA	1.435840192×10^{-2}
NVIDIA V100	FLUDA	1.435840192×10^{-2}
NVIDIA A100	FLUDA	1.435840192×10^{-2}
AMD MI210	FLUDA	1.435840192×10^{-2}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

GPU Implementation



FUN3D mini-app structure and porting workflow

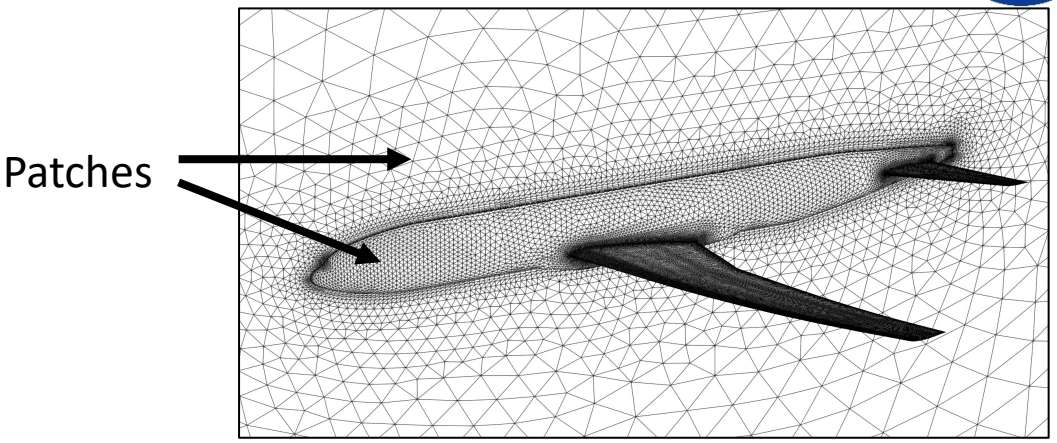
GPU Optimizations

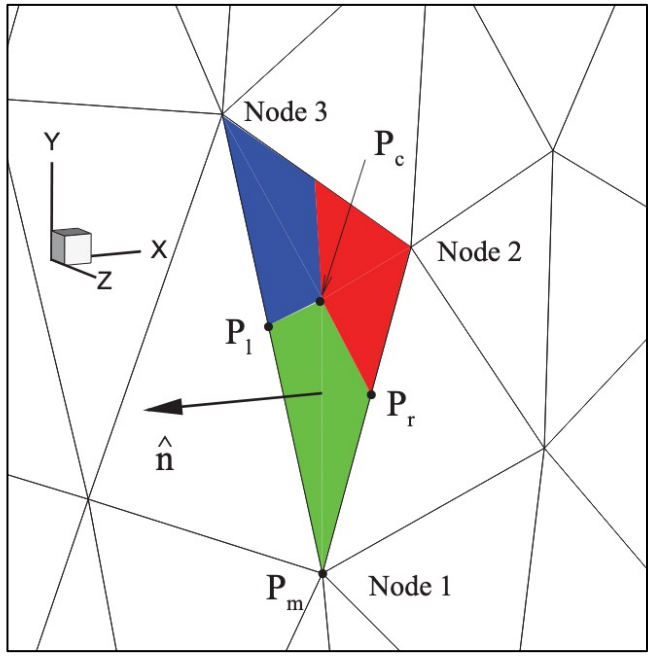


- Reduction of kernel state
  - Fortran implementation utilizes variable-length arrays (VLAs) for workspace
    - Since VLAs do not exist for C++, templating is extensively used
  - Initial naive CUDA port resulted in stack frames so large that the GPU ran out of memory immediately
  - To remedy this, we employ **hierarchical parallelism.** Multiple threads are assigned to a work item (such as a Jacobian) which reduces 2D arrays to scalars in many cases
  - Registers and shared memory are heavily used
  - Templates are used for almost every option to reduce registers and increase performance
- Reduce thread divergence
- Coalesced memory accesses
- Kernel launch parameter optimization
- GPU-aware MPI – e.g., Frontier MPI goes through GPUs directly and GPU-aware MPI is necessary
- See papers for more details

Parallelism Example [1/2]

- Boundary Jacobian Flux Kernel
  - Loop over boundary patches (NB)
    - Loop over triangles
      - Loop over nodes per triangle (3)
        - Compute flux Jacobian for BC type (NxN matrix)
      - Add contributions to global system
- Naïve Strategy
  - 1 kernel call per boundary patch
    - 1 thread per triangle (3*N*N Jacobian data per thread)
- Hierarchical Parallel Strategy
  - 1 kernel call per BC type for all boundary patches (all boundary patches at once in parallel)
    - 1 thread per vector of Jacobian (Nx1)
      - Each thread computes a Jacobian component of a node of a triangle
      - Nx1 Jacobian storage (15x less storage per thread for perfect gas)
      - **NB*3*N more parallelism (NB*15x for perfect gas)**
    - Templated on BC type; reduces registers from other BCs
    - **Boundary condition flux serial**
      - **Boundary conditions trivially extendable for non-GPU developers**
    - Parallel algorithm recovers original algorithm in serial and is performant compiled on CPU





NASA/TM-2011-217181

Parallelism Example [2/2]

- **Kernels must expose sufficient parallelism to attain high performance**
- Interior Jacobian example
- Loop over edges/dual-faces of grid
  - Construct NxN Jacobians
  - Add to global data
- 5-species, two-temperature gas model (N = 10)
- **Hierarchical parallelism is required for optimal GPU performance**

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
  }
}
```

Hierarchical parallel edge (dual-face) Jacobian kernel

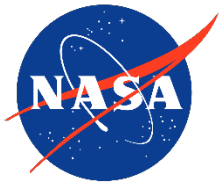
```
template<class CONFIG>
__GLOBAL__
void example_kernel_hierarchical_parallelism(...)
{
  constexpr int N = CONFIG::N;
  // local and shared variable declarations

  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
    }
  }
}
```

Architecture	Implementation	Speedup	Hardware MBW Ratio
Intel Skylake 6148 (40 cores)	Fortran	0.73	1.00
Intel Skylake 6148 (40 cores)	FLUDA Naïve	1.00	1.00
Intel Skylake 6148 (40 cores)	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

Hierarchical kernel 7.6x faster than naïve kernel on GPU, but only 1.1x slower on CPU

Performance scales with memory bandwidth ratio for optimal implementations

Multi-Architecture Approach



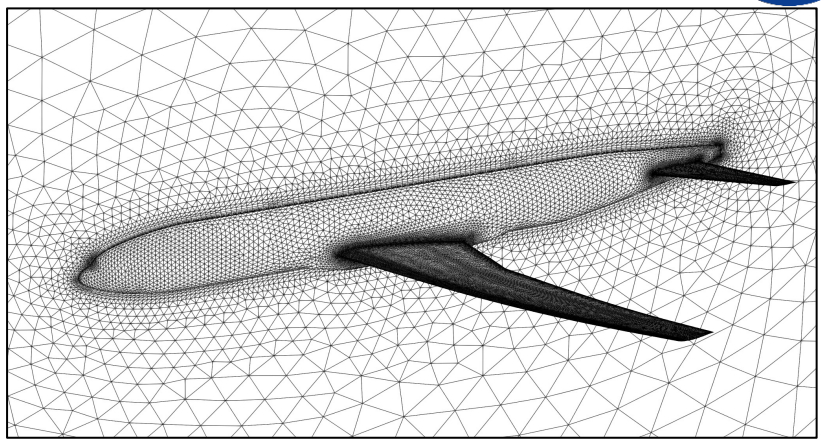
- Architecture-specific code used when employing hierarchical parallelism is much worse
- CPU and GPU code divergence is <2% of code base
- Linear solver (~50% of run time, <1% of code base) has architecture optimized implementations
- We achieve high percentage of peak memory bandwidth for linear solver (60-80% across architectures)
- Other key kernels are generally achieving 50% of peak or better according to profilers
- Performance across GPU architectures is obtained through autotuning of thread block parameters
  - Many kernels are templated on gas model (e.g., number of species) and element type
  - Each template combination has tuned threading parameters for each architecture

Results Overview

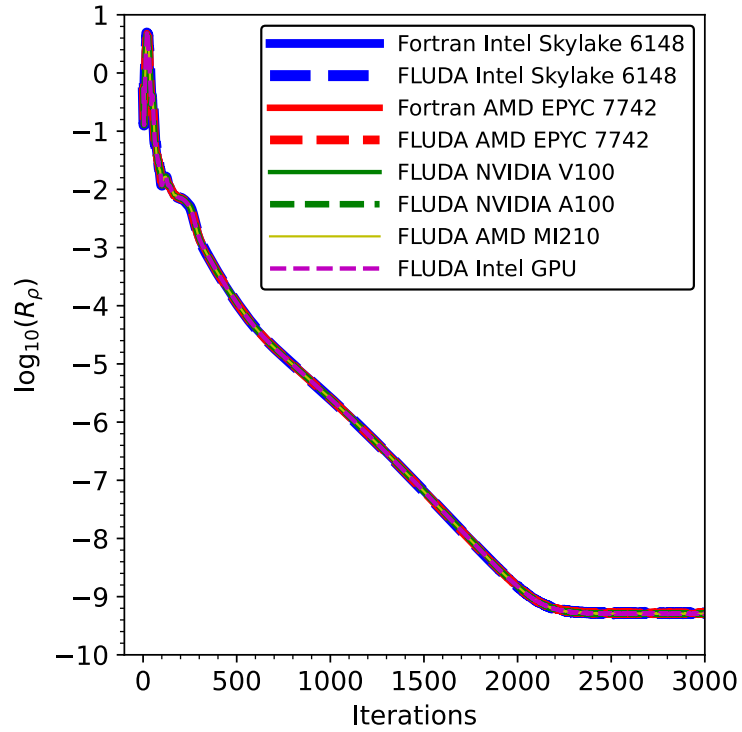
- Various results shown for a wide breadth of physics and applications
- Results are run mainly on NAS hardware
- Performance results are fastest times obtained after testing a variety of compilers and optimization flags
  - GPU-aware MPI is **not** used for these results on NAS
- RANS simulation of NASA Common Research Model (CRM) at Transonic Conditions
  - Correctness
  - Device-level performance
  - Representative of Cruise Conditions
- WMLES of High-Lift NASA CRM
  - Performance at scale and unsteady flow
  - Representative of Landing/Takeoff Conditions
- Hypersonic Crew Exploration Vehicle (CEV) and Sierra Space Dream Chaser© spaceplane
  - Performance of thermochemical nonequilibrium flows

RANS Simulation of the NASA CRM at Transonic Conditions

- Please see paper for details
- 3.7M points, 10M mixed elements
- SA-QCR2000, cruise condition
- Time to machine zero convergence
- **Hardware memory bandwidth (MBW) is theoretical and vendor-reported**
- **Using DPW2 and FUN3D in 2003 as baseline, FUN3D would take 62 hours (!) on 9 single-core 2.4 GHz Pentium 4 nodes with 2GB of 800 MHz memory for this case**
- **20 years later, we can now do the same simulation in a minute on a GPU node**



Architecture	Implementation	Time [min]	Speedup	Hardware MBW Ratio	Hardware MBW [GB/s]
Intel Skylake 6148 (40 cores)	Fortran	56	0.84	1.00	256
Intel Skylake 6148 (40 cores)	FLUDA	47	1.00	1.00	256
AMD EPYC 7742 (128 cores)	Fortran	29	1.62	1.60	409.6
AMD EPYC 7742 (128 cores)	FLUDA	27	1.72	1.60	409.6
NVIDIA 16 GB SXM V100	FLUDA	12	3.91	3.52	900
NVIDIA 40 GB SXM A100	FLUDA	7	6.55	6.05	1555
AMD MI210	FLUDA	10	4.55	6.40	1638.4

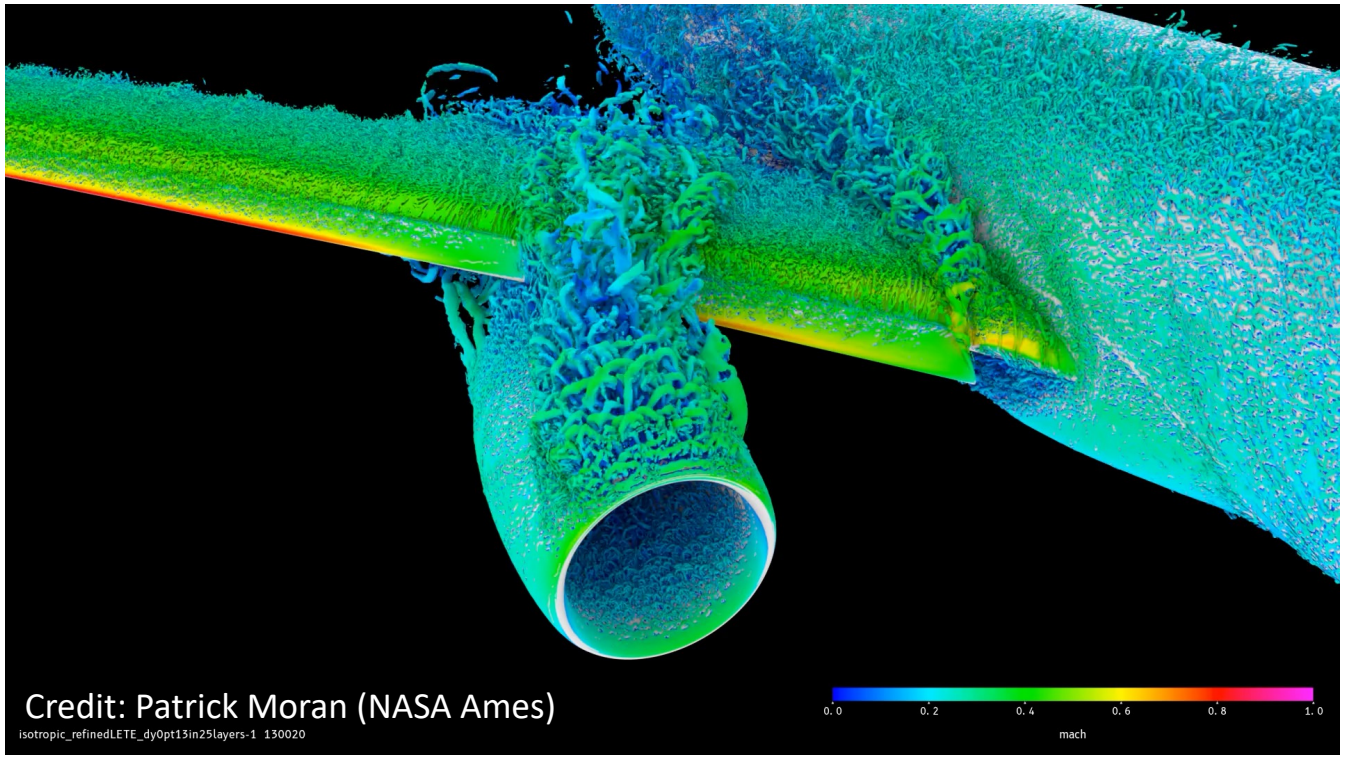


Speedup = Hardware device-level performance normalized to FLUDA CPU

Performance scales with memory bandwidth ratio

WMLES of High-Lift NASA CRM

- Please see paper for details
- Landing condition, a key design point
- 419M points, 812M mixed elements
- O(50) Convective Time Unit (CTU) for statistically stationary result



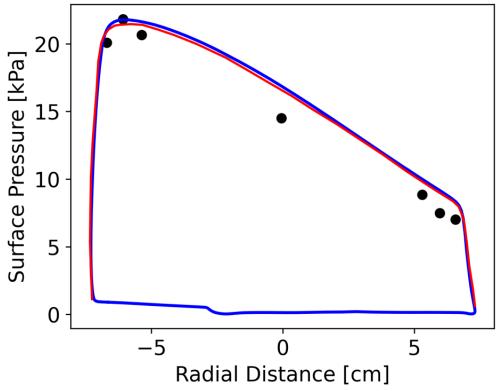
Speedup = Normalized hardware device-level performance
1 V100 ~ 2.13 EPYC nodes ~ 273 cores

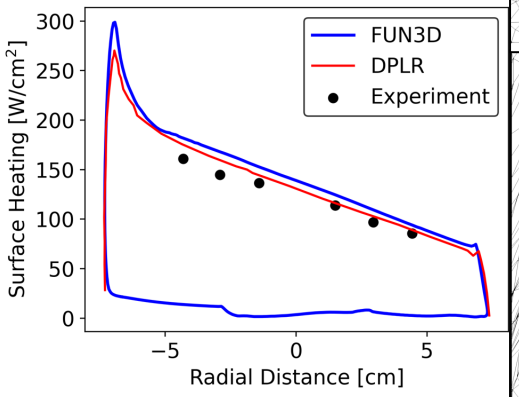
Architecture	Implementation	Time per CTU [min]	Speedup	Hardware MBW Ratio
200 AMD EPYC 7742 (25,600 cores)	Fortran	81	0.87	1.00
200 AMD EPYC 7742 (25,600 cores)	FLUDA	69	1.00	1.00
108 NVIDIA V100	FLUDA	60	2.13	2.20

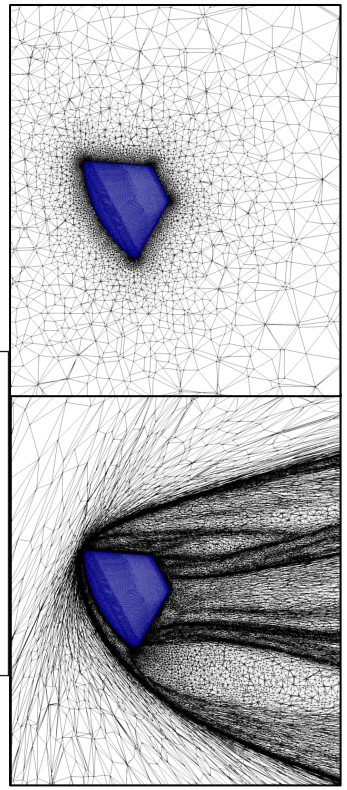
Performance scales with memory bandwidth ratio

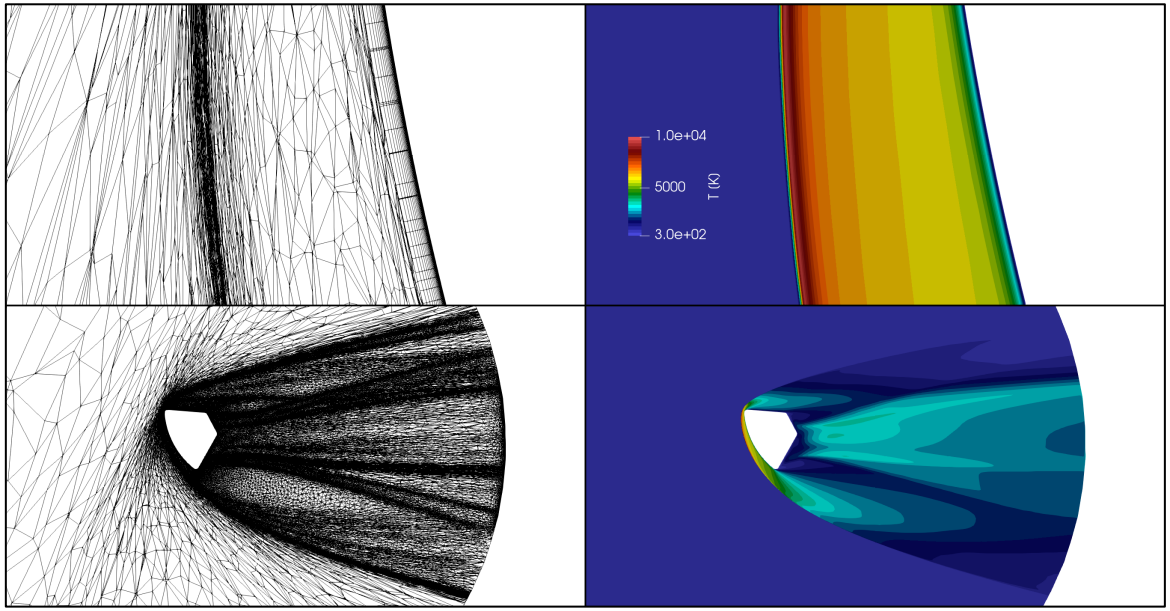
Crew Exploration Vehicle

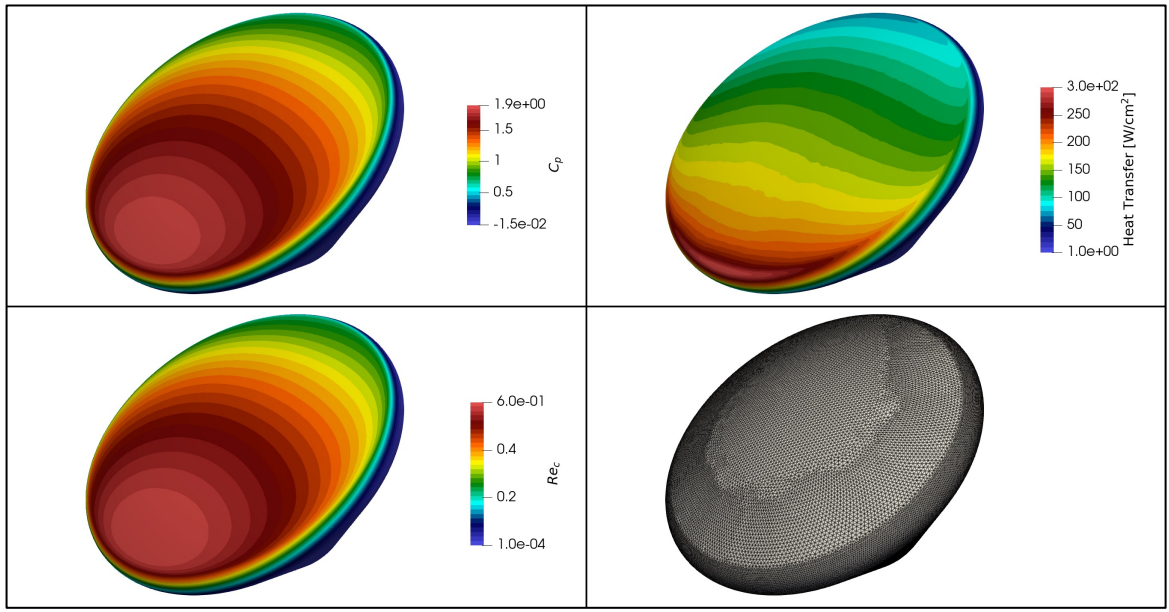
- Please see paper for details
- See past work for details on approach[1]
- FUN3D tutorial available online for this example[2]
- Fixed thin prismatic BL and surface grid, with tetrahedral adaptation using NASA *refine* elsewhere
- $u_\infty = 4.6 \frac{km}{s}, \alpha = 28°$
- Laminar flow, 5-species air, two-temperature model
- Comparisons to DPLR and experiment available
- 3M point, 12M mixed elements for final grid
- 15 CFD and refinement cycles









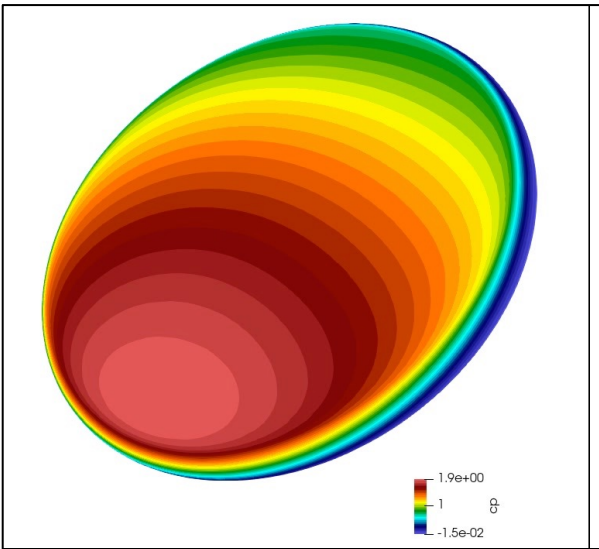


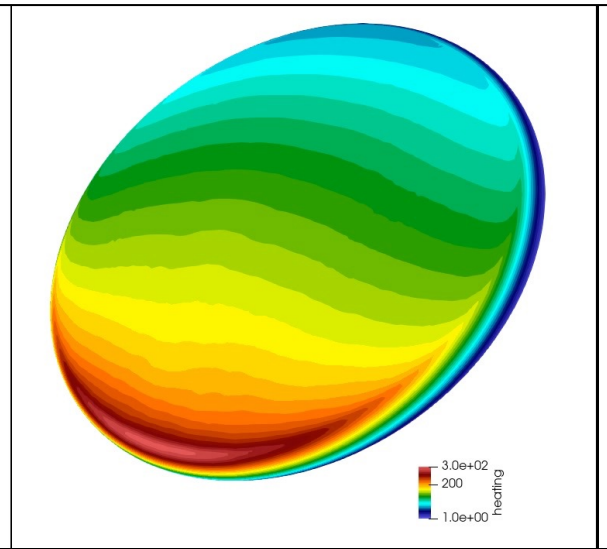
[1]Nastac, G., Tramel, R. W., & Nielsen, E. J. (2022). Improved Heat Transfer Prediction for High-Speed Flows over Blunt Bodies using Adaptive Mixed-Element Unstructured Grids. AIAA Paper 2022-0111.

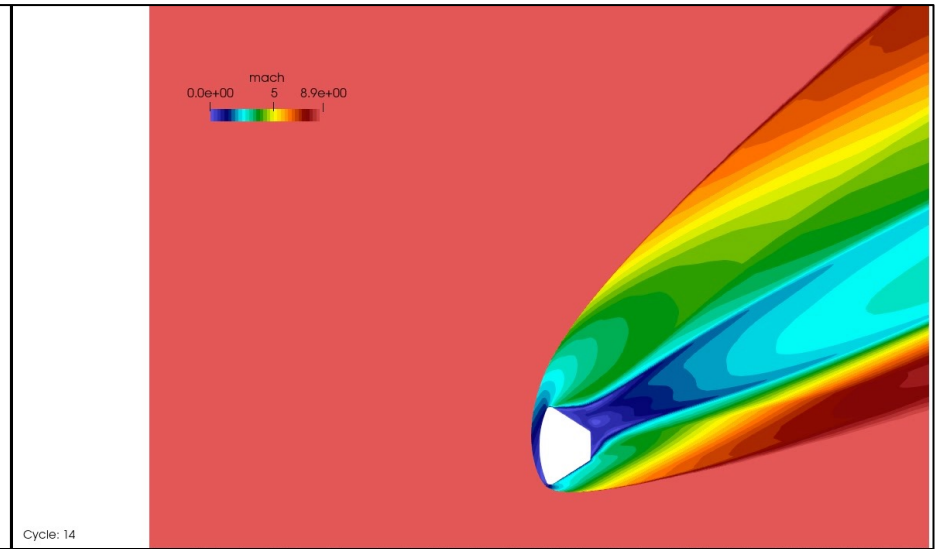
[2]https://fun3d.larc.nasa.gov/training-9.html#agenda_and_materials

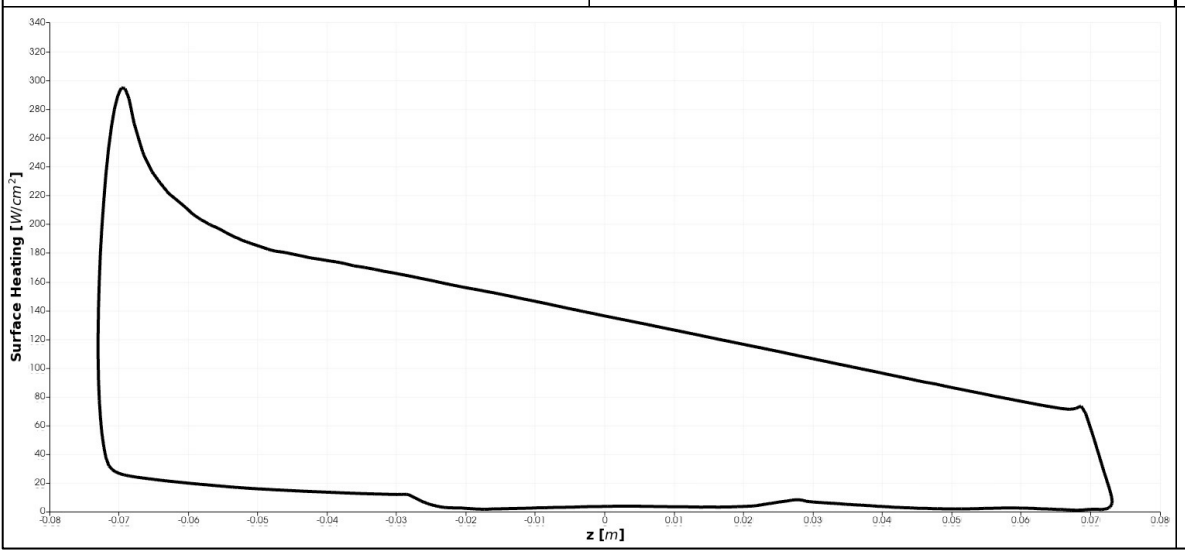
Crew Exploration Vehicle (cont.)

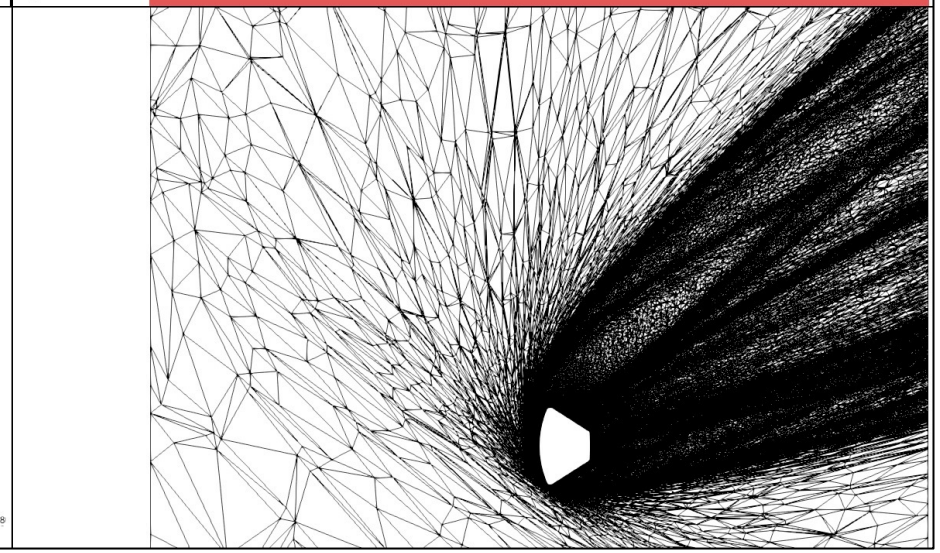
Example flow solution over time from FUN3D tutorial

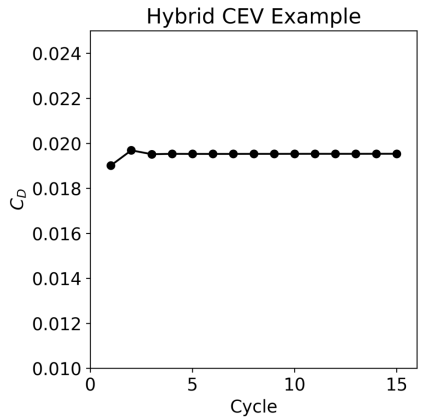




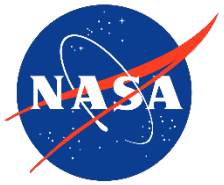








Crew Exploration Vehicle (cont.)



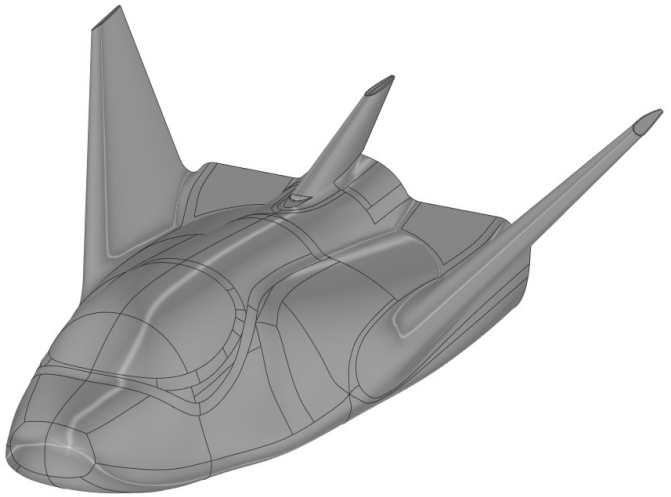
- NASA *refine* runs on CPUs and is used through files at this time
- I/O includes preprocessing and postprocessing
- GPU-enabled refinement will allow this approach to run more efficiently on GPU hardware
- A100 GPU results use 8 MPI ranks per GPU (generally 5-10% compute overhead while speeding up I/O)
- **This simulation is possible on a single node with one A100 GPU in under 6 hours**
  - Thermochemical nonequilibrium flow over a 3D capsule including wake and refinement with unstructured adapted grids

Architecture	Implementation	Total [min]	*refine* [min]	I/O [min]	CFD [min]	Speedup	Hardware MBW Ratio
15 AMD EPYC 7742 (1,920 cores)	Fortran	126	11	11	104	0.76	1.00
15 AMD EPYC 7742 (1,920 cores)	FLUDA	101	11	11	79	1.00	1.00
8 NVIDIA V100	FLUDA	162	82	21	59	2.51	2.20
4 NVIDIA 40 GB A100	FLUDA	150	57	14	79	3.75	3.80
1 NVIDIA 80 GB A100	FLUDA	337	57	18	262	4.52	4.72

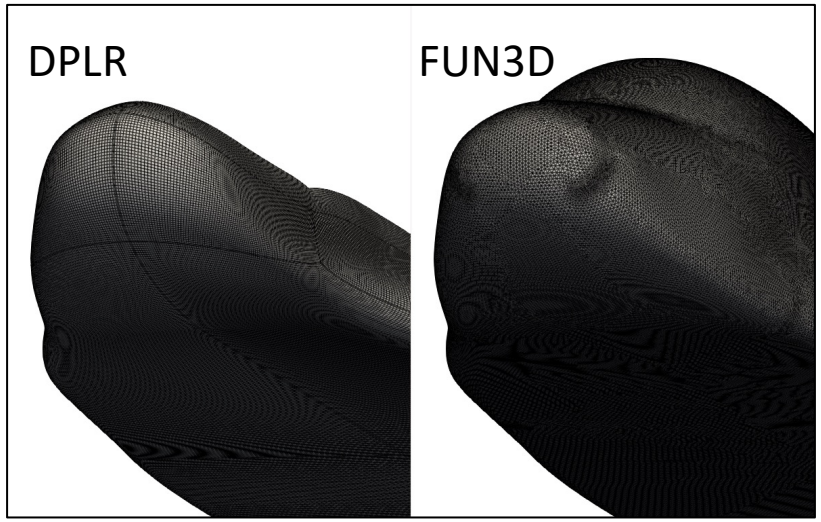
CFD performance scales with memory bandwidth ratio

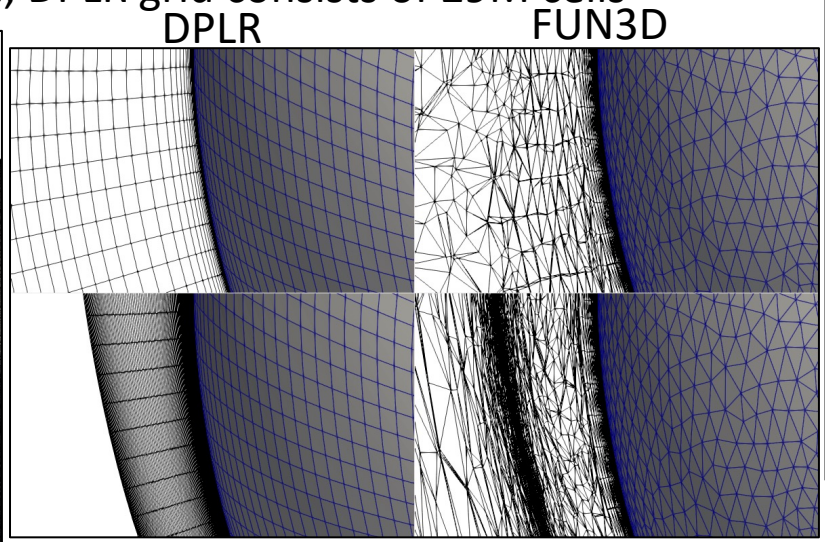
Speedup = Normalized hardware device-level performance
1 80 GB A100 ~ 4.52 EPYC nodes ~ 579 cores

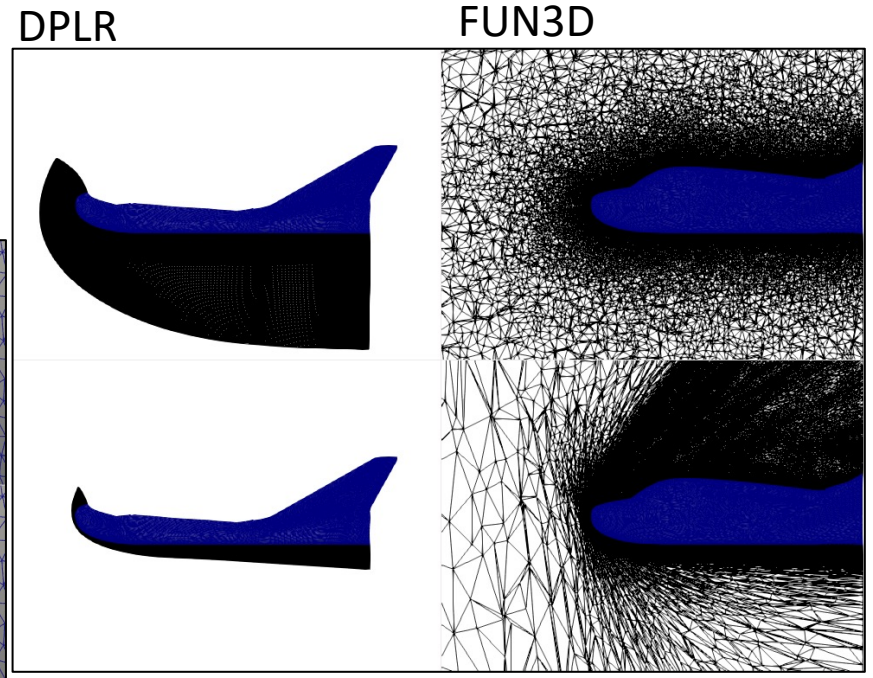
Sierra Space Dream Chaser



- Please see paper for details
- Fixed thin prismatic BL and surface grid, with tetrahedral adaptation using NASA *refine* elsewhere
- $u_\infty = 5.0\frac{km}{s}, \alpha = 40^\circ$
- Wall is modeled as reaction cured glass (RCG) coating and assumed in radiative thermal equilibrium
- Laminar flow, 5-species air, one-temperature model
- Final unstructured grid contains 12M points, 50M mixed elements
- 15 CFD and refinement cycles
- Comparisons to NASA DPLR available, DPLR grid consists of 29M cells

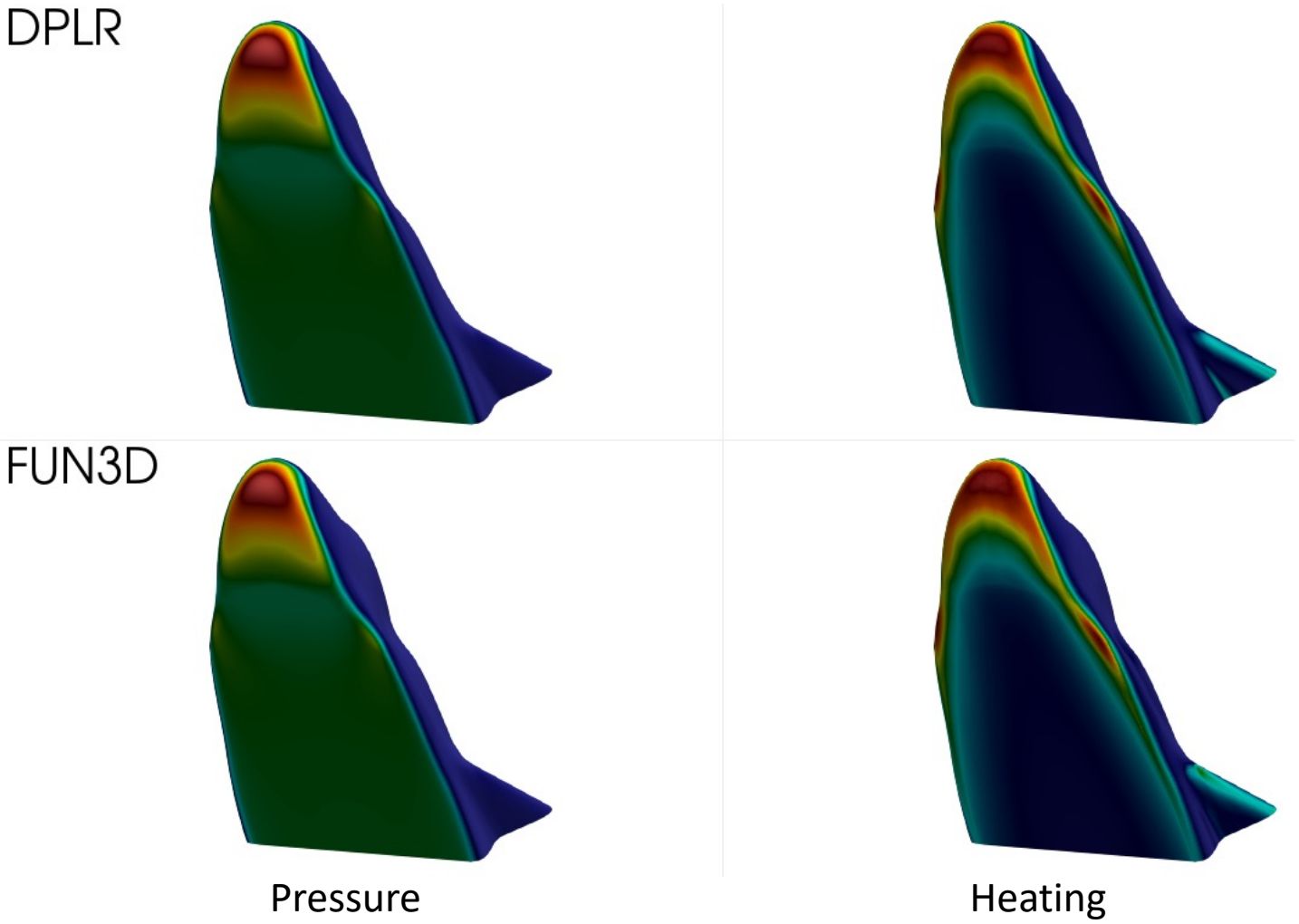


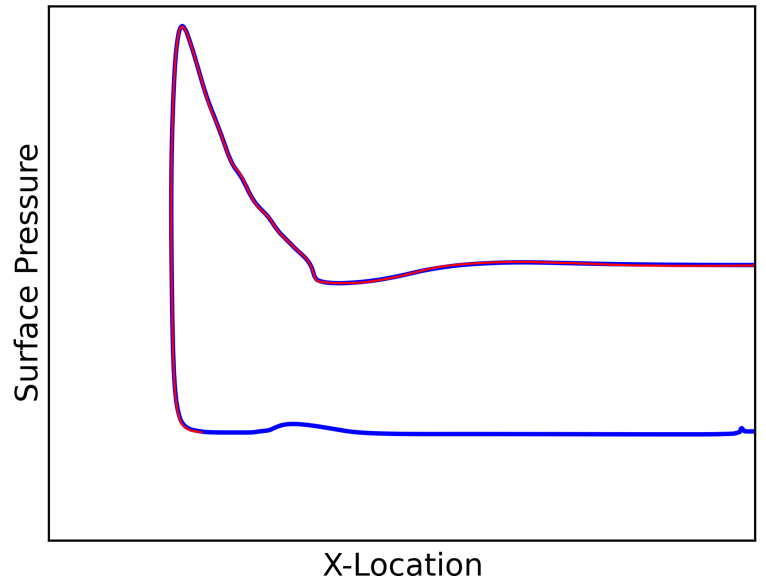


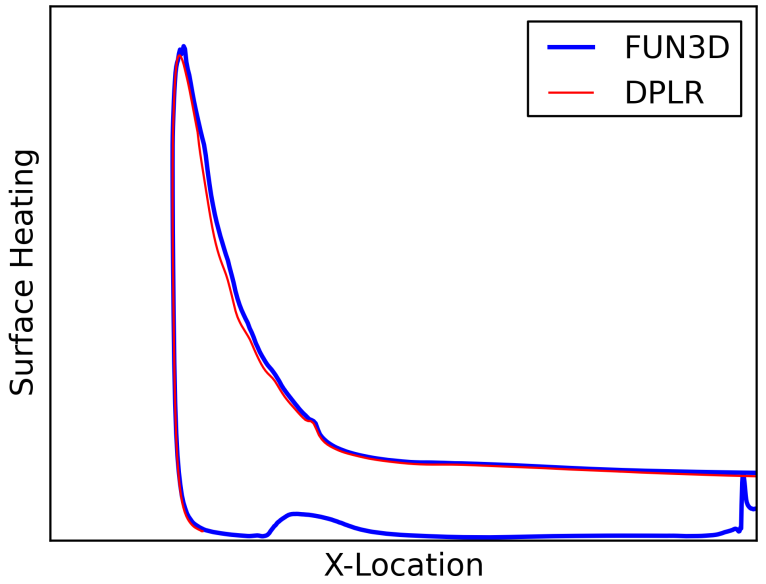


Credit: Sierra Space Corporation

Sierra Space Dream Chaser (cont.)







At stagnation point:
Pressure within 0.2%
Heating within 2.0%

Credit: Sierra Space Corporation

Sierra Space Dream Chaser (cont.)

- For GPU runs, grid refinement is a bottleneck due to running on CPU
- Current practice is to run refinement on separate CPU nodes, which complicates simulation process

Architecture	Implementation	Total [min]	*refine* [min]	I/O [min]	CFD [min]	Speedup	Hardware MBW Ratio
30 AMD EPYC 7742 (3,840 cores)	Fortran	132	24	7	101	0.83	1.00
30 AMD EPYC 7742 (3,840 cores)	FLUDA	115	24	7	84	1.00	1.00
16 NVIDIA V100	FLUDA	293	180	39	74	2.13	2.20

Speedup = Normalized hardware device-level performance
1 V100 ~ 2.13 EPYC nodes ~ 273 cores

CFD performance scales with memory bandwidth ratio

Code	DOFs	Iterations	refinement [min]	CFD [min]	Total [min]
DPLR	28.6M	7,000	6	52	58
FLUDA CPU	12.2M	21,000	24	91	115

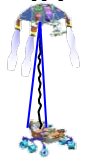
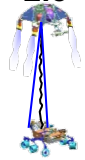
FUN3D speed is comparable to DPLR for these simulations

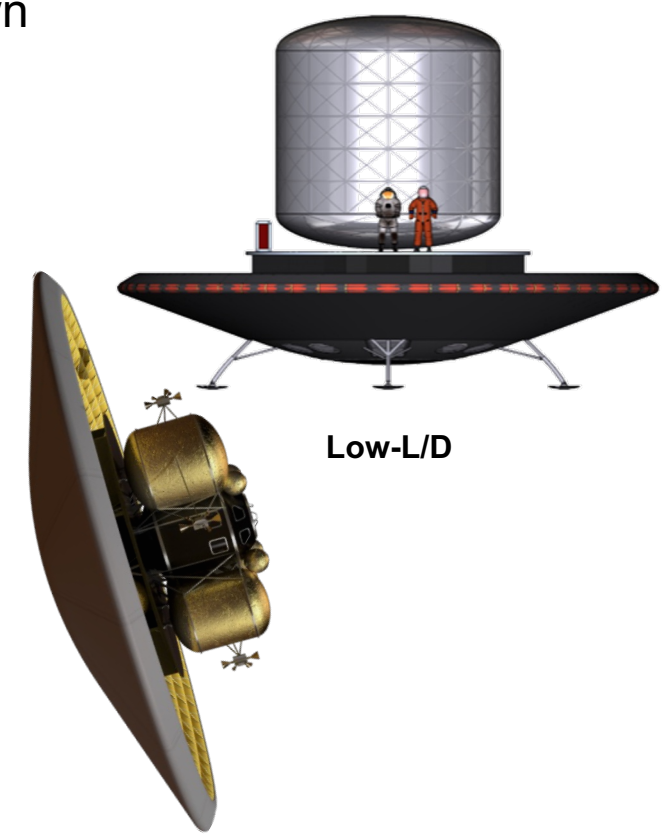
Credit: Sierra Space Corporation

Supersonic Retropropulsion Trajectory Simulations

Retropropulsion for Human Mars Exploration

- Human-scale Mars landers require new approaches to all phases of Entry, Descent, and Landing
- Cannot use heritage, low-L/D rigid capsules → deployable hypersonic decelerators or mid-L/D rigid aeroshells
- Cannot use parachutes → retropropulsion, from supersonic conditions to touchdown
- No viable alternative to an extended, retropropulsive phase of flight
- Limited understanding / numerous questions; physical testing is infeasible

	Viking	Pathfinder	MERs	Phoenix	MSL	InSight	M2020	Human-Scale Lander (Projected)
Entry Capsule (to scale)								
Diameter (m)	3.505	2.65	2.65	2.65	4.52	2.65	4.5	16 - 19
Entry Mass (t)	0.930	0.584	0.832	0.573	3.153	0.608	3.440	40 - 65
Parachute Diameter (m)	16.0	12.5	14.0	11.8	19.7	11.8	21.5	N/A
Parachute Deploy (Mach)	1.1	1.57	1.77	1.65	1.75	1.66	1.75	N/A
Landed Mass (t)	0.603	0.360	0.539	0.364	0.899	0.375	1.050	26 - 36
Landing Altitude (km)	-3.5	-2.5	-1.4	-4.1	-4.4	-2.6	-2.5	+/- 2.0
Landing Technology	 Retro-propulsion	 Airbags	 Airbags	 Retro-propulsion	 Skycrane	 Retro-propulsion	 Skycrane	Retro-propulsion



Low-L/D

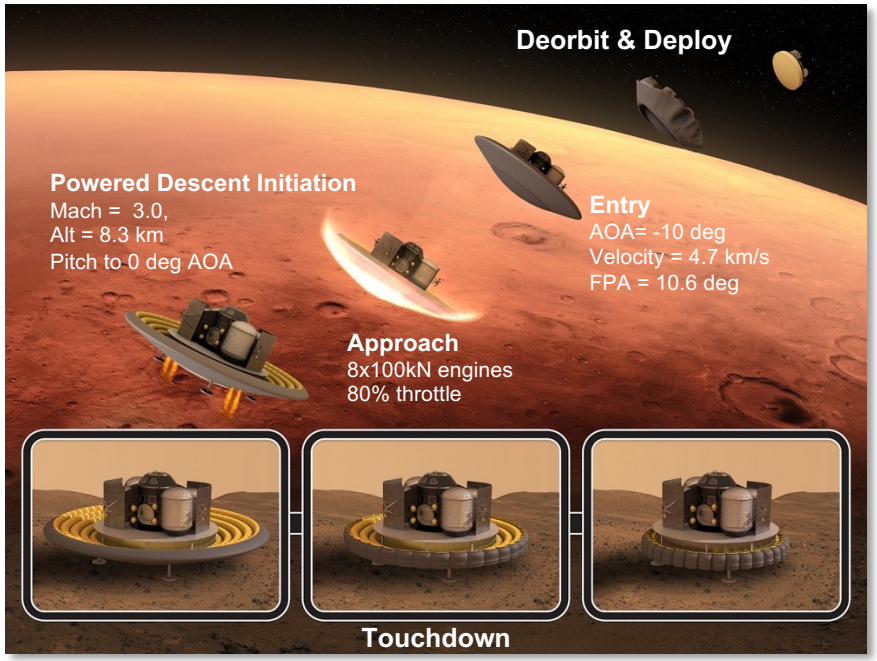
Steady progression of "in family" EDL

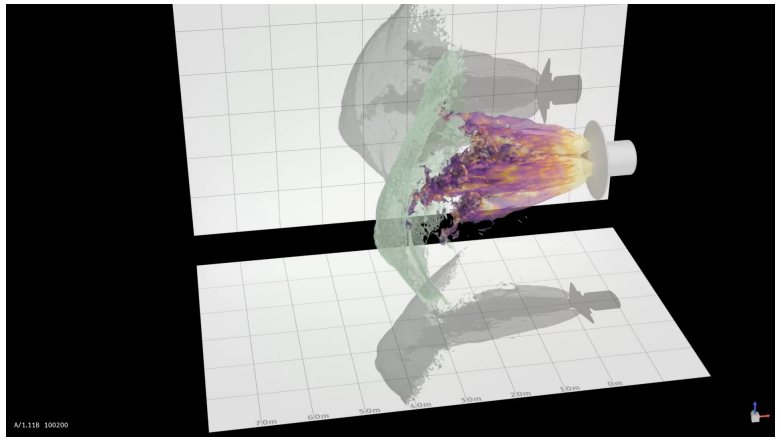
New EDL Paradigm

2018-2021 Summit Campaigns

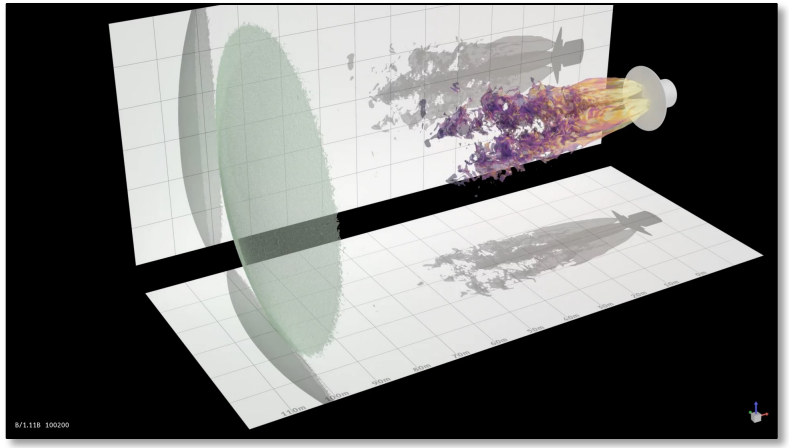
Retropropulsion for Human-Scale Mars Landers

- LOX/CH_4 rocket engines in Martian CO_2 atmosphere
- CFD static simulations of 10 species/19 reactions (15 PDEs) on grids of ~7 billion elements
- Broad range of spatial and temporal scales
- Game-changing computational performance: Two-day runs simulating seconds of real-time on 16,000 NVIDIA V100 GPUs
  - Equivalent of several million CPU cores (billions of CPU core-hours total)
- Would require system of 100,000 CPU cores for 2 months to do one run
- Big data:
  - 90 GB of asynchronous I/O every 30 seconds for two days yields ~1 PB/run
  - 60 TB migrated from ORNL to NASA Ames daily

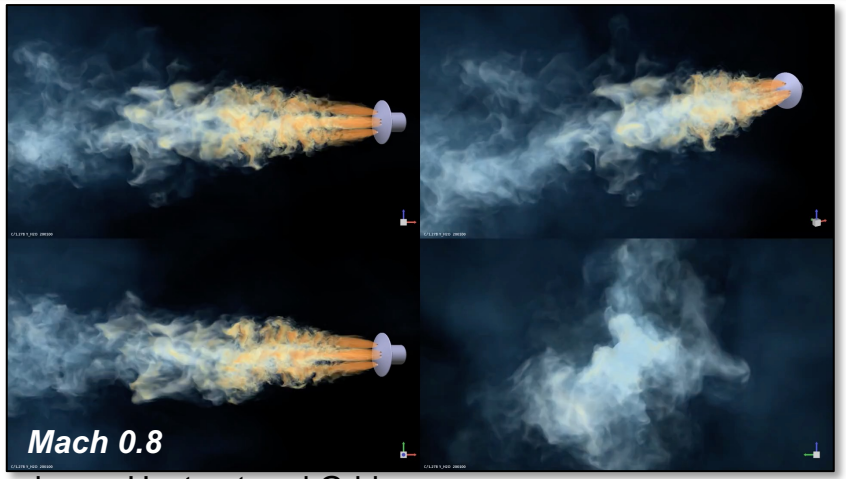




Mach 2.4
Dynamic Turbulent Flowfield



Mach 1.4
~100 m Shock Standoff Distance



CFD-Based Trajectory Analysis

- Current design of aerospace vehicles, including these Mars landers, relies on Monte Carlo trajectory simulations which utilize aerodynamic databases that are commonly generated using RANS CFD simulations
- While these approaches have been sufficient for Mars EDL in the past, recent technologies for human exploration such as retropropulsion must improve tools
- Unsteady methods such as URANS, DES, and LES better predict flow phenomena such as turbulence and separation
  - Increase cost of databases by over an order of magnitude
- Unsteady methods do not necessarily greatly reduce uncertainties
  - Ignore hysteresis effects
- Latest static simulations show phenomena lasting several seconds of physical time which would not occur in a real trajectory
- **CFD-based trajectory analysis is a grand challenge problem recently posed to the aerospace CFD community**
  - **Desired for both aeronautics and space applications (and combined)**
  - **Augment static simulations by truly "flying" high-fidelity, physics-based trajectories coupled with flight mechanics algorithms**
  - **Enables testing of flight mechanics algorithms in a high-fidelity simulation environment**
- The incoming generation of exascale systems enables us to consider such an approach with **scale-resolving methods**

Typical CPU-Based Costs for Space Launch System Database Generation

Database	Code	Solutions (Grid Size)	Wall Time
Ascent	FUN3D	1,380 (60M)	2-4 weeks
Ascent	OVERFLOW	1,000 (500M)	2-3 months
F & M Wind Tunnel	FUN3D	600 (40M)	1 week
Booster Separation	FUN3D	13,780	3 months
Booster Separation	Cart3D	25,000	3 months

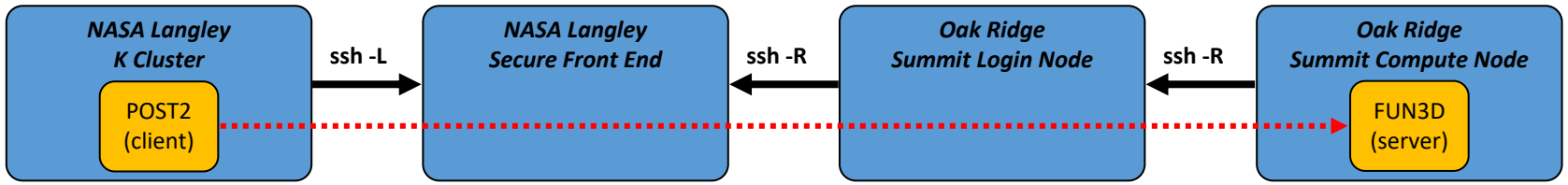
Derek Dalle, NASA ARC

Program to Optimize Simulated Trajectories II (POST2)

- **Flight-validated, generalized, event-based, point-mass vehicle & trajectory simulation codebase**
  - 3/6/Multi-DOF
  - Continuously developed and maintained in-house at LaRC
  - “Developers-as-Users”
- **Key Features**
  - Custom, robust input language
  - Interfaces with user-provided multidisciplinary engineering models and flight software
  - Built-in trajectory optimization
  - API permits external tools (e.g., Copernicus) to call POST2 and optimize on it directly (new for 2023!)
- **Key Applications**
  - Statistical analysis of end-to-end integrated performance
  - Orbital & atmospheric trajectory optimization & design
  - GN&C algorithm development & assessment
  - Off-nominal, faults, aborts, and margin analysis
- **For the Current Effort**
  - Leveraging POST2 provides far more capability than a traditional 6-DOF implementation within the CFD solver
  - FUN3D provides high-fidelity aerodynamics to POST2

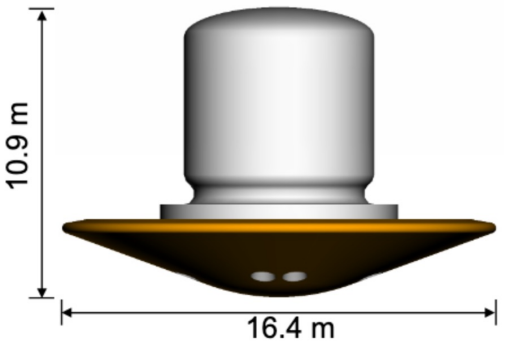
FUN3D-POST2 Coupling

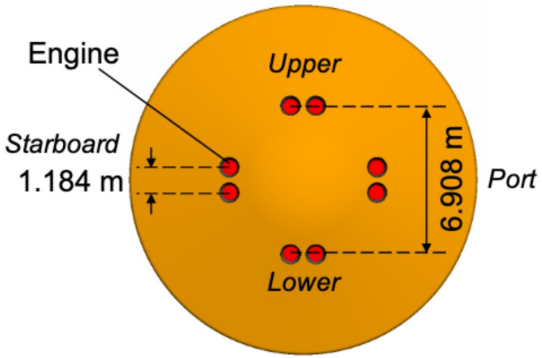
- FUN3D and POST2 are coupled through a socket-based interface
- Since POST2 is ITAR-restricted, it is executed on a dedicated node located at NASA, while FUN3D executes at scale on Summit at Oak Ridge
  - POST2 provides the FUN3D server with the current vehicle position, orientation, and throttle settings, while also requesting the current aerodynamic loads from the server
  - FUN3D receives the POST2 inputs and responds with the aerodynamic loads
- The timestep for the two applications differs by a factor of ~150; exchanges occur every minute
- One roundtrip costs ~0.5 seconds, including POST2 execution



Computational Overview

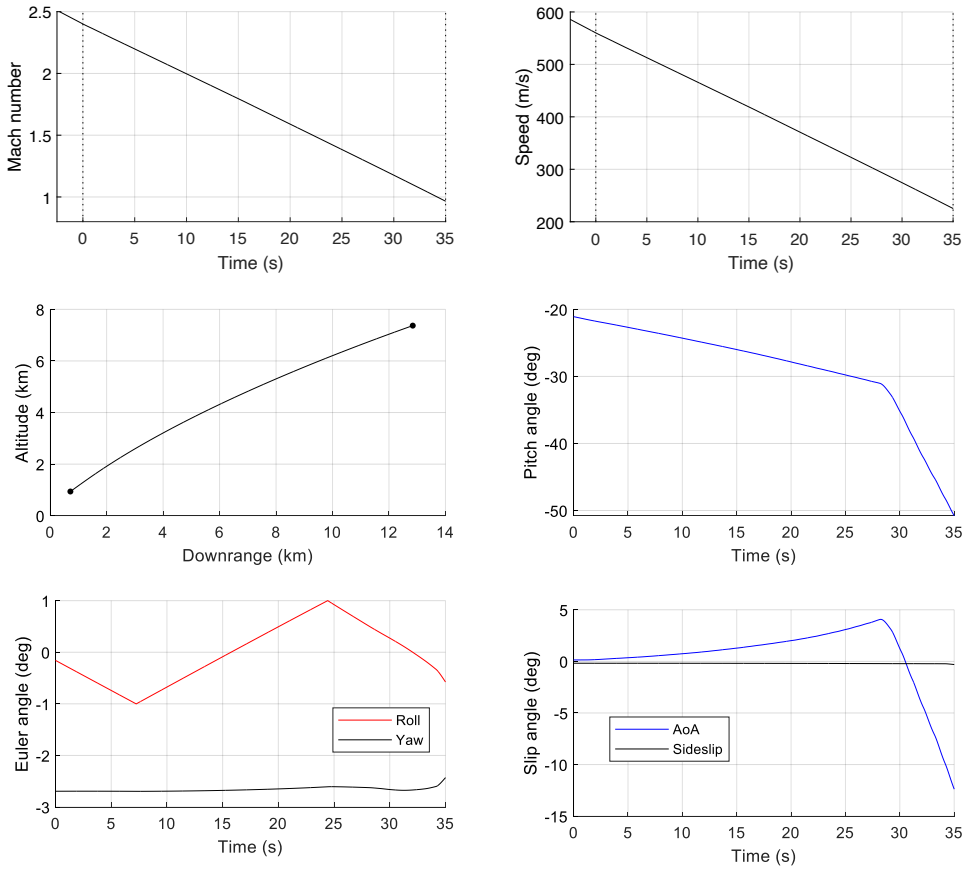
- Domain dimensions in *kilometers* with the ability to resolve flow features on the order of centimeters
- CFD grid consists of ~1.3 billion points and ~7 billion elements
- Walls resolved with $y^+ \approx 1$
- Detached-Eddy Simulation (DES) turbulence model
- Perfect-gas model based on CO_2 is used to reduce computational cost
- A quasi steady-state solution at Mach 2.4 static conditions and 80% main engine throttles is used to initialize the simulation
- **Atmospheric conditions are approximated with a constant density and temperature**
- Three flights of progressive complexity run during campaign
  - Prescribed 1DOF axial deceleration
  - Prescribed 6DOF
  - **Constrained two-way 6DOF – focus of the work**
    - Vehicle axial force is exchanged with POST2
    - POST2 models are used for other forces and moments
    - POST2 propulsion model (including RCS) is used for control
    - Throttle control and RCS not currently modeled in FUN3D, although available
- CFD is run on 5,532 NVIDIA V100s on Summit
  - Wall time per time step is about half a second
- Physical time step for the implicit CFD is 67 μs (~450-1200 steps to traverse 16-meter heat shield); POST2 time step is 10 ms
  - Since the CFD time step is considerably smaller, a curve fit is applied to the POST2 data being transmitted to FUN3D
- **Runtime is approximately 3 days for 35 seconds of physical time**

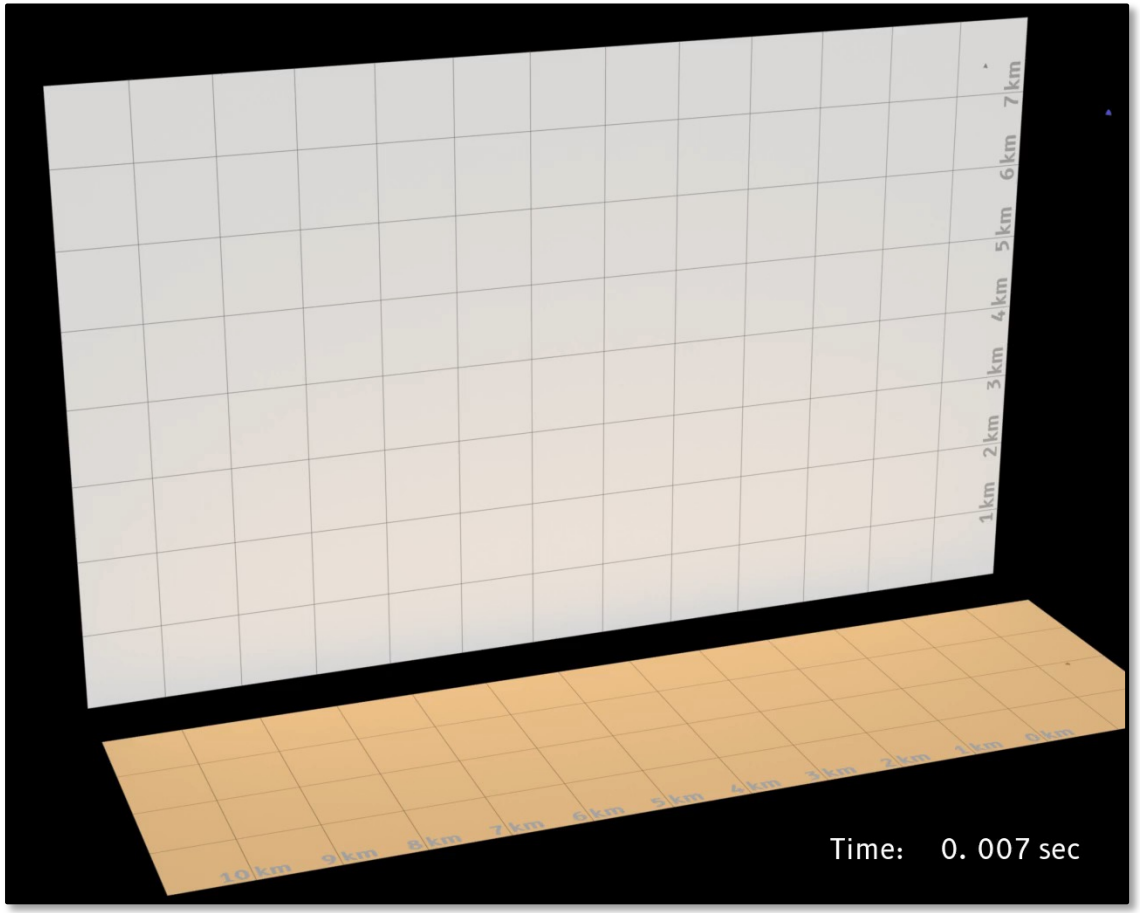




Two-way 6DOF Trajectory

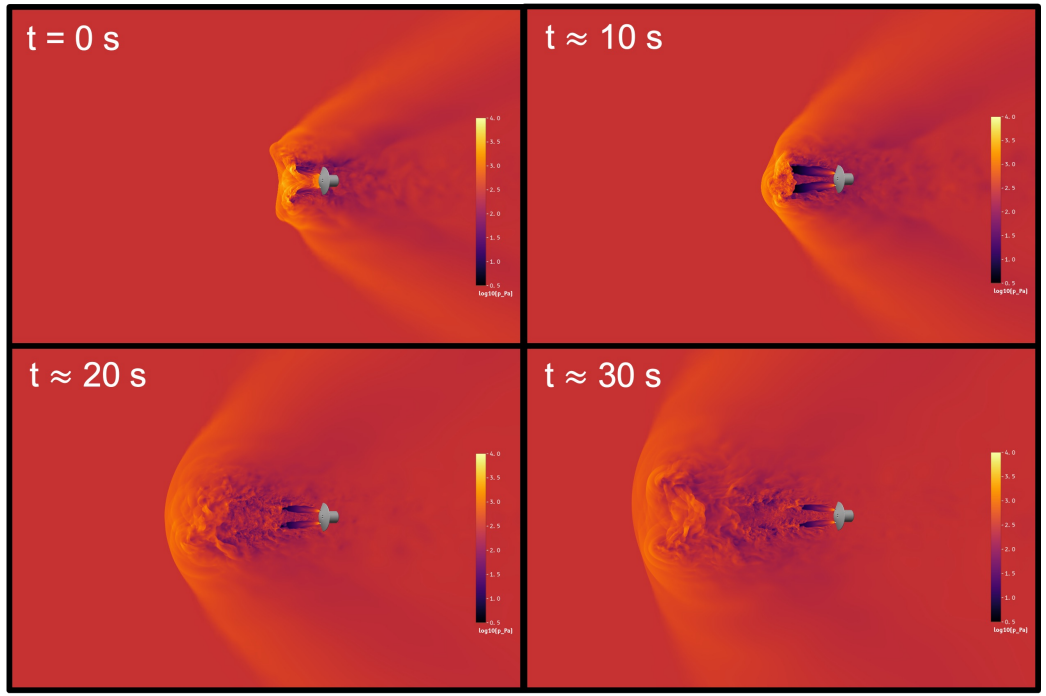
- The 6DOF trajectory of interest is a 35-second segment starting immediately after main engine startup at mid-supersonic conditions
- During this time span, the vehicle decelerates from approximately Mach 2.4 to just under Mach 1 as it approaches the landing phase





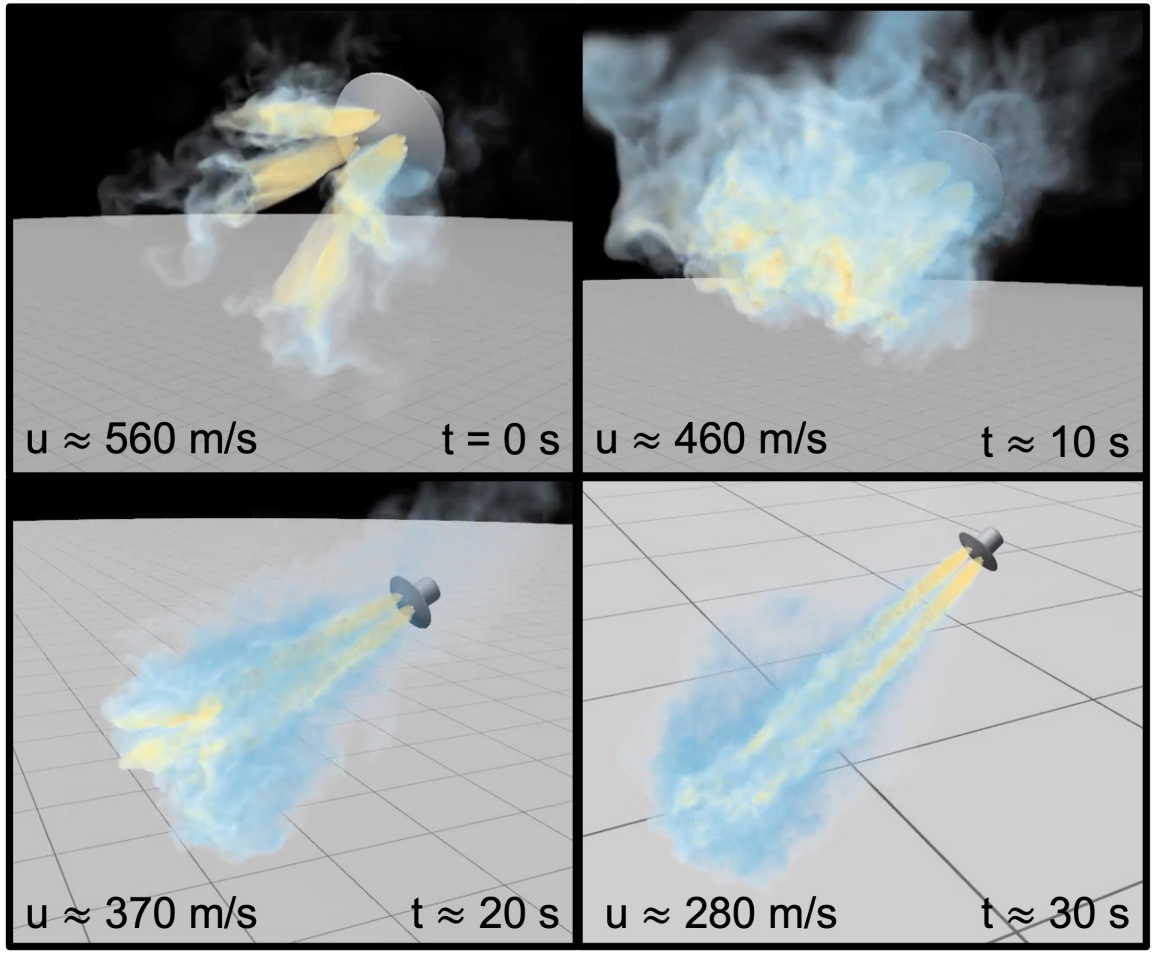
Results

- Instantaneous snapshots shown every 10 seconds
- POST2 dictates 6DOF
- Engine plumes grow to 100+ meters at the final subsonic conditions; flowfields qualitatively similar to static simulations performed in prior campaigns



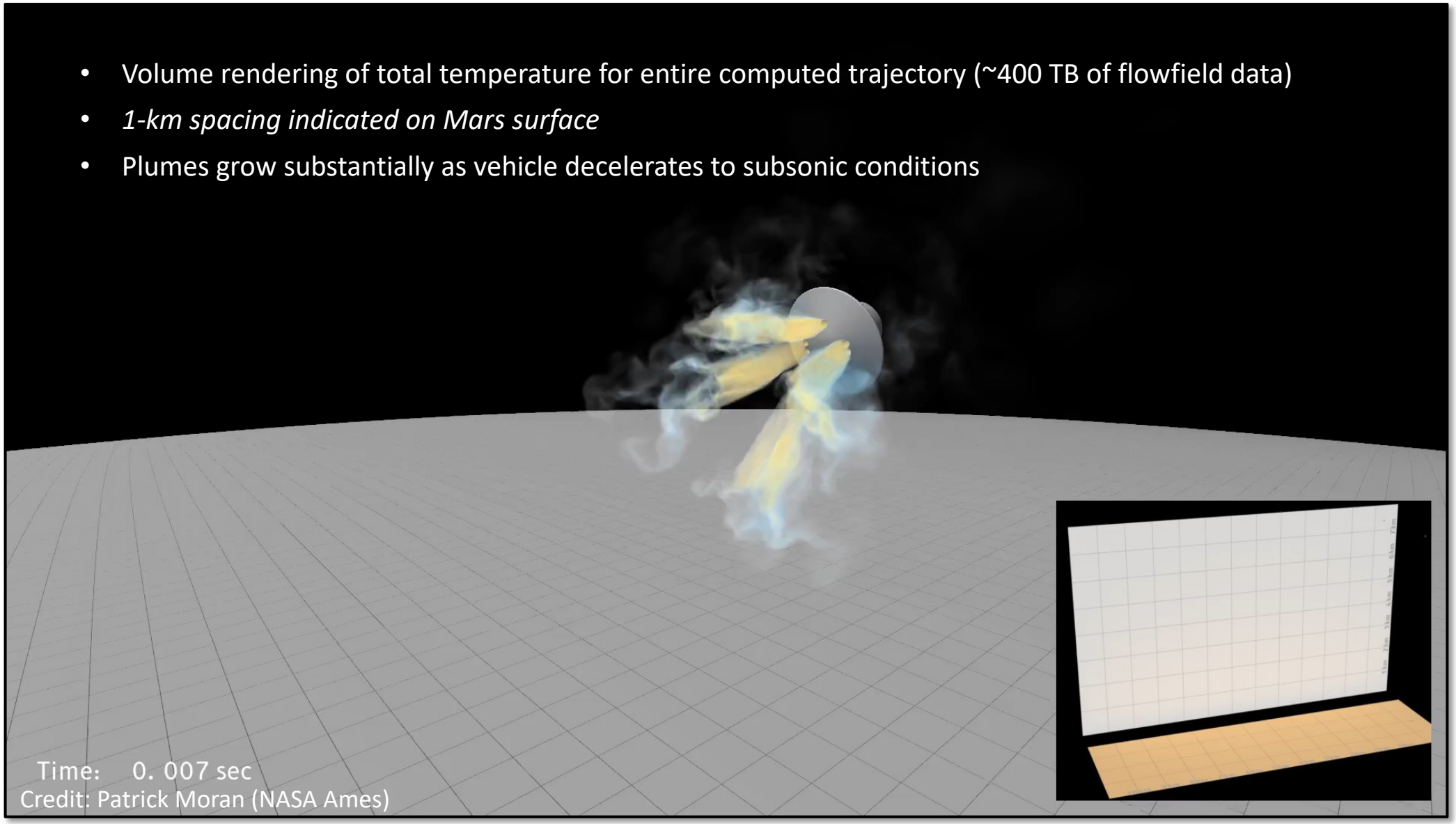
Pressure distributions across the trajectory

Total temperature at 10-second intervals (1-km spacing indicated on Mars surface)



Results

- Volume rendering of total temperature for entire computed trajectory (~400 TB of flowfield data)
- *1-km spacing indicated on Mars surface*
- Plumes grow substantially as vehicle decelerates to subsonic conditions



Summary

- Demonstrated multi-architecture approach for implicit CFD on unstructured grids on problems across speed regime
- Thin abstraction layer similar to CUDA C++ and HIP employed
- GPU-optimized kernels run efficiently on CPUs with a single thread
- Performance scales with memory bandwidth to first order as expected
- **Performant GPU-enabled CFD software enables faster and cheaper design cycles for those with desktop or cluster resources versus CPU-enabled CFD software**
- Perfect Gas RANS simulations on 4M point grids in minutes on 1 GPU

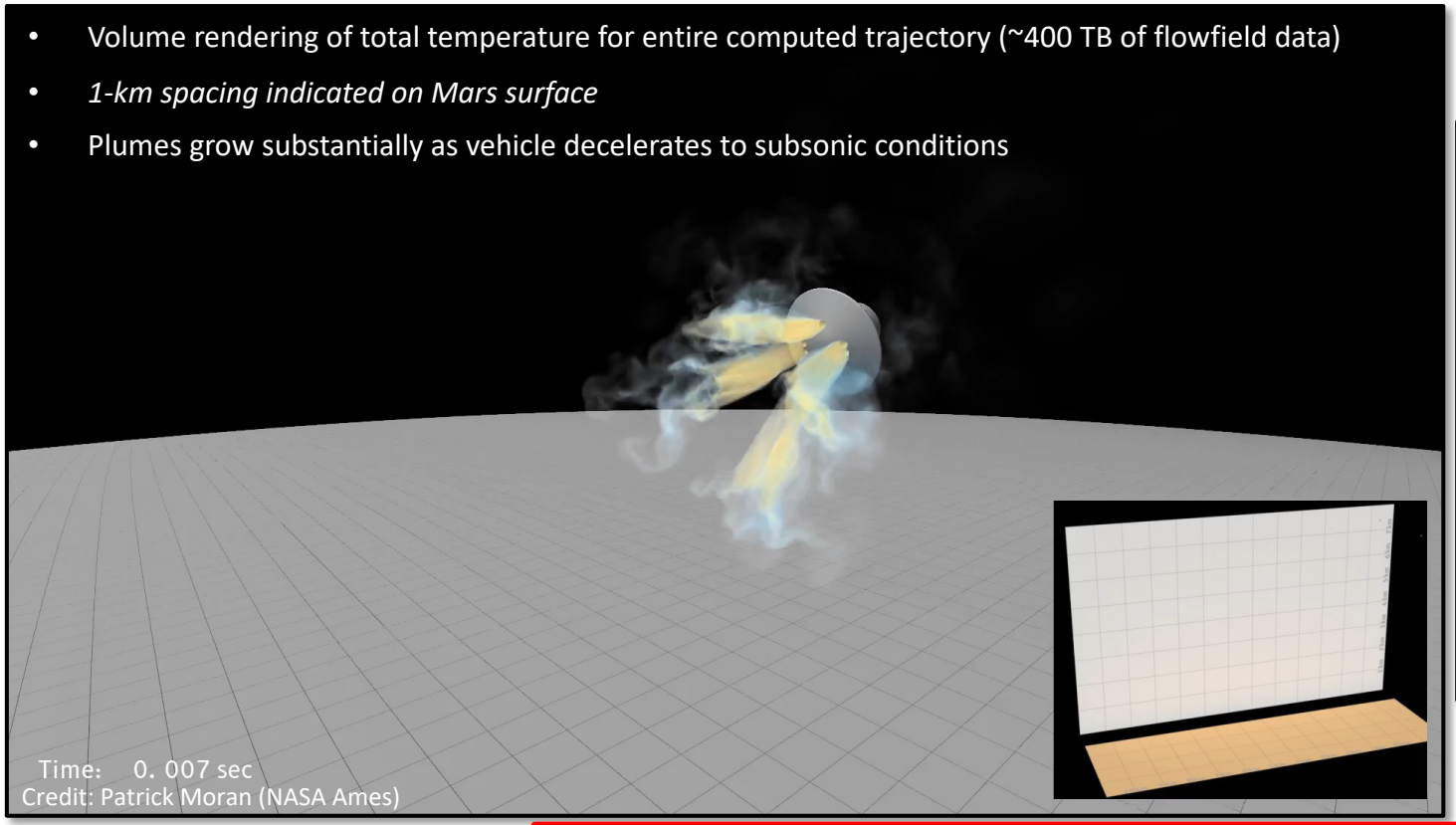
Future Work

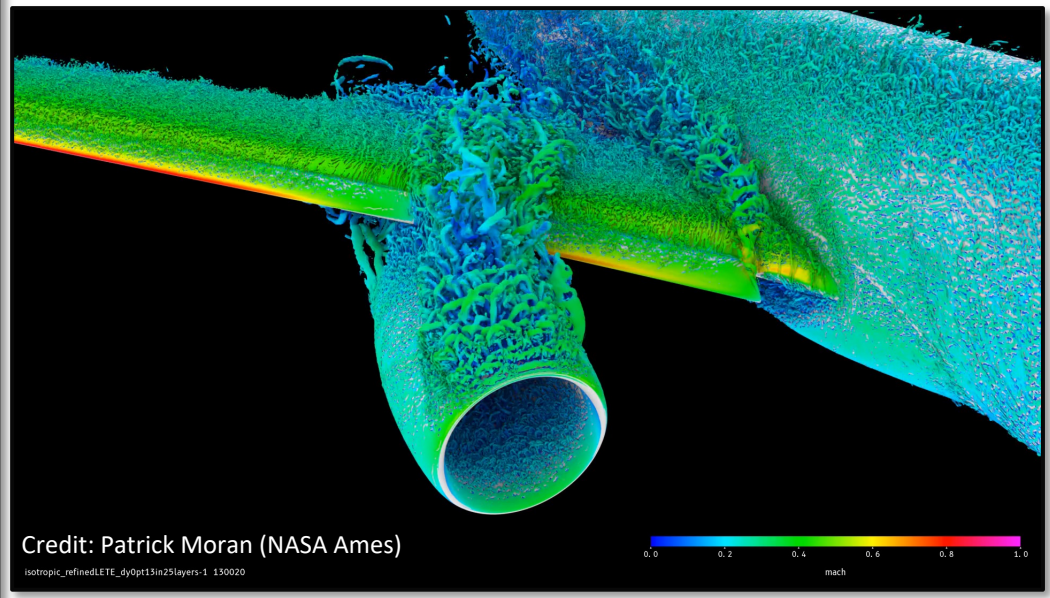
- Further adapt CPU capabilities for multi-architectures: pre/post processing, grid adaptation
- “Digital-twin” Trajectories/Simulations

Questions?

Thanks to all of our collaborators, partners, and sponsors that have helped make this happen!

- Volume rendering of total temperature for entire computed trajectory (~400 TB of flowfield data)
- *1-km spacing indicated on Mars surface*
- Plumes grow substantially as vehicle decelerates to subsonic conditions





Looking to bring on new team members – we are hiring!
Eric.J.Nielsen@nasa.gov

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.