AIAA SciTech Forum
6-10 January 2025, Orlando, FL
AIAA SCITECH 2025 Forum

10.2514/6.2025-0784

GPU Acceleration of a Two-Dimensional Adjoint-Enabled Real Gas Hypersonic Flow Solver

Andrew C. Kirby*
School of Computing, University of Wyoming, Laramie, WY 82071, USA

Dimitri J. Mavriplis†
Scientific Simulations, LLC., Steamboat Springs, CO 80487, USA

This paper presents the GPU acceleration of a legacy Fortran-based two-dimensional viscous Navier-Stokes solver designed for aerothermodynamic applications. The code utilizes a multi-species, two-temperature model for chemically reacting gases and employs a line-implicit solver on unstructured meshes. Additionally, the solver includes the exact discrete adjoint system of the real-gas model for uncertainty quantification. To ensure hardware vendor neutrality, the OCCA abstraction model is employed for adapting the legacy codebase to GPU architectures.

I. Introduction

Porting legacy Computational Fluid Dynamics (CFD) solvers for Aeronautical and Astronautical Engineering applications, commonly written in the Fortran language family, for leveraging Graphical Processing Units (GPU) presents challenges but is necessary for leveraging state-of-the-art computing power. These legacy solvers primarily rely on distributed-memory parallelism for performance on CPU-centric architectures. Modern GPUs, however, are heavily based on shared-memory parallel architectures, thus require a different programming paradigm. Some key challenges for porting these Fortran-based solvers include addressing Fortran's relatively limited support for GPU integration, restructuring legacy code to optimize memory usage and data access patterns for GPU execution, and ensuring performance portability across different GPU vendors.

The most common GPU-vendor-agnostic solutions for this purpose include high-level directive-based approaches such as OpenACC [1, 2] or OpenMP [3, 4], which facilitate incremental porting with minimal modifications to the original Fortran codebase. Lower-level abstraction via explicit parallelism-based solutions like the Kokkos [5, 6] or RAJA [7, 8] programming models enable more flexible, performance-portable designs, allowing the same codebase to target accelerator architectures effectively, but require developers to write parallel constructs in the code directly using library-provided abstractions.

In directive-based approaches, obtaining computational performance can rely heavily on the compiler's capabilities to interpret the directives and generate efficient parallel code for the target architecture (e.g., GPUs or CPUs). This reliance on "compiler magic" forces a tradeoff between performance and code modification. In contrast, the explicit parallelism abstraction frameworks Kokkos and RAJA are C++-based libraries that require converting the legacy Fortran source code compute kernels into C++ lambda functions to express parallelism. This intrusive approach leads to substantial developer effort, which can be error-prone.

In this paper, we turn to the Open Computing Language for Accelerators (OCCA) [9] thread-programming abstraction model, an open-source library for employing GPUs, CPUs, and FPGAs to achieve performance portability. OCCA allows developers to efficiently write performance-portable code in the OCCA Kernel Language (OKL) [10], an intermediate representation (IR) for explicit shared-memory parallelism. This IR is just-in-time compiled into various back-ends, including serial code, CUDA, HIP, DPC++, OpenMP, and OpenCL, enabling NVIDIA, AMD, and Intel GPU devices with a single codebase. OCCA demonstrated efficient and scalable implementations of multiple applications [11–14], including the Department of Energy (DOE) incompressible CFD solver Nek5000 [15] – branded NekRS [16] – as part of the DOE's Exascale Computing Program's libCEED [17] project. By adopting such strategies, developers can modernize their solvers to achieve significant performance improvements while maintaining code maintainability and cross-platform compatibility.

We demonstrate the efficient conversion of a legacy Fortran codebase to utilize GPUs with explicit parallelism, enabling high performance using OCCA. First, we introduce the legacy flow solver, including the problem motivation, and detail the implementation of the numerical discretization for the physics models. We then discuss the approaches to accelerate the software and present the computational performance achieved. Lastly, we provide a roadmap for efficiently porting legacy Fortran codes.

*AIAA Member; Corresponding Author (akirby@uwyo.edu)
†AIAA Fellow

II. Background

The successful design of next-generation planetary entry vehicles relies on the accurate prediction of their flight stability properties during entry into the atmosphere. During this flight phase, the flow field around the vehicle is characterized by a strong bow shock that causes chemical and thermal nonequilibrium, boundary layer transition, and separation [18]. Reproducing these flow conditions in ground-based experimental facilities is challenging, while in-flight tests are limited and expensive. CFD presents a viable alternative to the different experimental approaches for predicting entry, descent, and landing aerodynamics. As such, the codebase providing the foundation for this work is an unstructured mesh CFD solver designed to compute the solution of the two-dimensional Navier-Stokes equations for a real gas in chemical and thermal nonequilibrium developed by Lockwood and Mavriplis [19, 20], detailed hereafter.

A. Governing Equations and Numerical Discretization

The numerical solution of high-enthalpy, hypersonic flows requires the solution of the multi-species, multi-temperature Navier-Stokes (NS) equations around the entry vehicle. In vector form, these equations are stated:

$$\frac{\partial \mathbf{Q}}{\partial t} + \nabla \cdot \mathbf{F}_{inv}\left(\mathbf{Q}\right) = \nabla \cdot \mathbf{F}_{visc}\left(\mathbf{Q}\right) + \mathbf{S}\left(\mathbf{Q}\right) \tag{1}$$

These equations describe the conservation of the total mass, momentum and energy of the gas mixture, of the chemical species that compose the reacting gas, and of the translational and vibrational energy of the gas mixture. The variable $\mathbf{Q}$ is the vector of conserved variables that contains the total mass, momentum and energy, the mass fraction of the chemical species of the gas, and the vibrational temperature, while the terms $\mathbf{F}_{inv}\left(\mathbf{Q}\right)$ and $\mathbf{F}_{visc}\left(\mathbf{Q}\right)$ are the inviscid and viscous fluxes, respectively. Finally, the source term $\mathbf{S}\left(\mathbf{Q}\right)$ contains the chemical reactions that occur among the components of the gas, and its numerical evaluation is computationally expensive and introduces stiffness into the system of equations.

The computational cost of the evaluation of the source term is a consequence of the large number of chemical reactions that need to be evaluated even for a limited number of species in the gas mixture. Since the number of chemical reactions may grow quadratically with the number of species, the computation of the source term can become a dominant component of the overall cost of the simulation when larger numbers of species are considered. Additionally, since the characteristic time scales of the chemical reactions are faster than the times scales associated with the conservation of the total quantities, the computation of the finite-rate chemistry of the gas mixture introduces stiffness in the coupled system of equations. Furthermore, the accurate computation of mechanical and thermal loads acting on the entry vehicle requires the resolution of both the thermal and viscous boundary layers, typically accomplished with the use of small cells, clustered near the vehicle surface. However, cell clustering introduces additional stiffness and stability constraints, complicating the numerical solution of the discretized NS equations even further.

Non-linear equation discretization

To overcome these difficulties, fully-implicit time-integration methods have been widely adopted for the numerical solution of hypersonic, chemically-reacting flows [21–24]. Focusing on steady-state problems, the discretized NS equations for a fully-implicit method become

$$\frac{\mathbf{Q}^{n+1} - \mathbf{Q}^{n}}{\Delta\tau} + \mathbf{R}\left(\mathbf{Q}^{n+1}\right) = \mathbf{0} \tag{2}$$

where $\Delta\tau$ is a pseudo time used to overcome the initial transient, and n is the current iteration or pseudo-time level. The array $\mathbf{R}\left(\mathbf{Q}^{n+1}\right)$ is the residual vector that contains the discretized inviscid and viscous fluxes, and the source term. The discretized non-linear system of equations is then solved using Newton's method. The method requires the solution of a linear system of equations at every pseudo time

$$\left[\frac{I}{\Delta\tau} + \frac{\partial \mathbf{R}}{\partial \mathbf{Q}}\right] \Delta\mathbf{Q}^{n+1} = -\mathbf{R}\left(\mathbf{Q}^{n}\right) \tag{3}$$

where the block-sparse matrix $\frac{\partial \mathbf{R}}{\partial \mathbf{Q}}$ is the Jacobian of the discretized residual vector. Line-implicit solution strategies combined with a low-order approximation of the Jacobian are employed to solve the linear system [22, 24, 25]. Originally developed for structured grids, line-implicit methods add implicitness in the direction of strong coupling in regions of high mesh anisotropy. For unstructured grids, line structures are typically identified by grouping together cells in the direction normal to the boundary-layer surface, although different line-extraction strategies are available. A line ends when no more cells can be added to the line and cells that cannot be grouped into any existing line are organized in lines of zero length. A block-tridiagonal solver is then employed to solve the linear system, and the approach naturally reverts to a regular point-implicit solver for lines of zero length.

More recent solution strategies have focused on ensuring faster and deeper convergence of the non-linear system. To this end, the Newton iteration is combined with preconditioned Krylov-based methods for the solution of the linearized system of equations [26, 27]. Since Krylov methods only require the action of the Jacobian onto a vector, modern automatic differentiation techniques can be employed to compute the matrix-vector products, reducing development time and storage requirements and enabling quadratic convergence of the non-linear system. Preconditioning of the Krylov method is used to accelerate convergence of the linear system. The Jacobian of a lower-order discretization is typically used as the preconditioning matrix, and the line-implicit Jacobi solver previously described can be used to solve the preconditioning system.

Spatial discretization

The implemented real gas model consists of a 5-species, 2-temperature model for non-ionizing air, easily extensible to include more species (e.g., an 11-species model incorporating ionization). The solver implements both the Dunn-Kang [28] and Park [29] models for chemical kinetics within the 2D code. A cell-centered finite volume discretization based on the AUSM+UP flux function [30] is employed for the inviscid flux calculation with a frozen speed of sound approach to enable real-gas flows [31] and a Green-Gauss gradient reconstruction strategy [32], limited using a combination of a pressure switch [33] and the smooth Van-Alabada limiter [34–36], provides second-order spatial accuracy. Transport properties, such as viscosity, thermal conductivity, and diffusion coefficients, are calculated based on linear interpolation of tabulated collision integrals.

The discretized non-linear equations are solved in a fully coupled fashion, using various approaches, including the point-implicit and line-implicit solvers and a Newton-Krylov method, which employs the point/line solver as a preconditioner. The point- and line-implicit solvers employ the Jacobian of the van Leer-Hanel flux function [37] to improve the diagonal dominance of the linear system. The generalized minimal residual method (GMRES) uses the exact linearization of the discretized residual to accelerate convergence to machine zero. A typical non-linear solution strategy consists of overcoming the initial transients using the line-implicit solver for a fixed number of iterations, then activating the Newton-Krylov method once the initial transients resolve. Furthermore, the aerothermodynamic solver is augmented with its discrete adjoint linearization to perform uncertainty quantification of chemically reacting flows [20].

B. Hypersonic Benchmark

The test case explored for this work is the hypersonic benchmark on a cylinder geometry found in the FUN3D and LAURA Manuals [21]. The grids utilized in this study are supplied from LAURA and are publicly available on the FUN3D website. Table 1 lists the flow conditions for this test case.

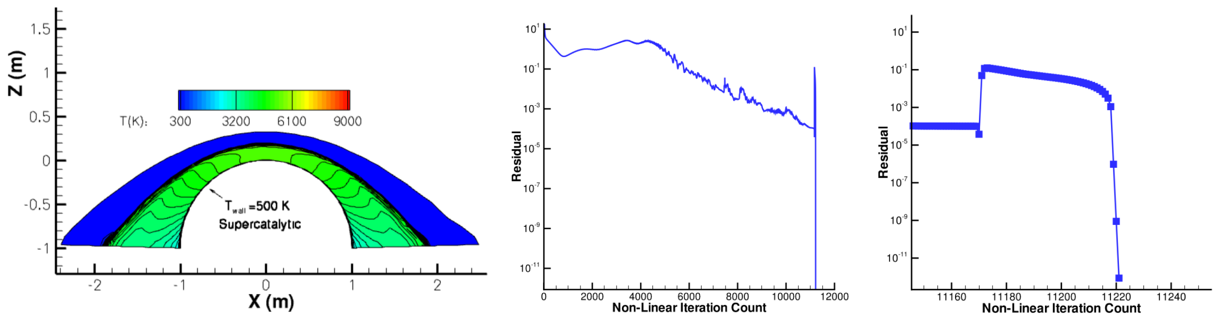
Table 1 Hypersonic Benchmark Flow Conditions

Parameter	V_∞	ρ_∞	T_∞	T_{wall}	M_∞	Re_∞	Pr_∞
Value	5 km/s	0.001 kg/m^3	200 K	500 K	17.605	376,960	0.72

The grids employed in this work are structured meshes with their statistics listed in Table 2. Figure 1 – reproduced from [19] – shows the temperature contours computed for a 5 km/s flow over a circular cylinder with a super-catalytic, fixed-temperature wall along with the convergence history in terms of density residual. Here the flow is started impulsively and run initially with the line solver. After several orders of magnitude reduction in the residual are achieved, the solution is restarted using the preconditioned GMRES solver, which is seen to be capable of converging the solution to machine precision. Figure 2 illustrates Mach and temperature profiles on the **ex10** mesh produced by the GPU-based implementation presented in this paper.

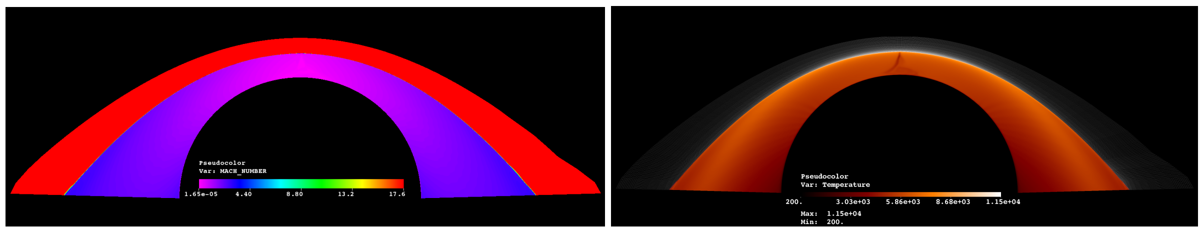
Table 2 Hypersonic Benchmark Mesh Definitions

Mesh	**Elements**	**Lines**	**Elements per Line**
ex05	3,840	60	64
ex06	138,240	360	384
ex10	384,000	600	640
ex20	1,536,000	1,200	1,280



(a) Temperature (b) Convergence History (c) Convergence Zoom

Fig. 1 Flow solution and convergence history computed with the aerothermodynamic CPU solver for the hypersonic benchmark case (reproduced from ref. [19]).



(a) Mach Number (b) Temperature

Fig. 2 Hypersonic benchmark solutions computed on the ex10 mesh using the GPU solver.

III. Computational Methodology

This project aims to create a vendor-agnostic GPU implementation of a legacy CFD solver written in 90,000 lines of Fortran 90. In addition, we strive to establish an effective development workflow for systematically and efficiently porting Fortran codebases while achieving high performance on GPU devices.

The *efficient use* of developers' time is essential, as building such a workflow could be transferable to significantly larger scientific projects supported by public funds. For instance, many important legacy scientific applications are also written in Fortran, including much of NASA's CFD software and the Department of Energy's climate modeling software. For example, the Energy Exascale Earth System Model (E^3SM) [38] contains over 1.8 million lines of Fortran code distributed across more than 4,100 files. Equally important, a systematic approach to porting code is critical to help reduce errors in the development process. Here, we detail our experiences of porting the codebase using a directive-based approach first, then turning to vendor-specific language for explicit shared-memory parallelism, and finally arriving at a vendor-agnostic approach with explicit parallelism and maintaining the same *look* of the Fortran kernel implementations.

A. Abstraction-based Programming Models

Our initial efforts focused on using the directive-based OpenACC framework, specifically designed for GPU computing, by providing a high-level abstraction for offloading code to GPUs. OpenACC provides backend support for NVIDIA and AMD GPUs. Developers annotate code with "`#pragma acc parallel`" or "`#pragma acc kernels`" to specify loops or regions to be executed on the GPU, while OpenACC automatically handles data transfer between host and device, though developers can optimize this further by indicating critical data regions using "`#pragma acc data`." This approach allows developers to directly modify the source code but places heavy reliance on the compiler to produce quality parallelization. We first profiled the CPU-based implementation of the transport properties kernel used in the source term for the chemical reactions, shown in Listing 8 in Appendix A, which previously implemented OpenMP shared-memory parallelism for CPU-based architectures. Leveraging these legacy parallelism efforts, the OpenMP directives were replaced by OpenACC statements. However, this approach quickly demonstrated insufficient performance without significantly manipulating the source code structure and data layout. Profiling the code performance on an NVIDIA 32GB V100 GPU revealed the results in Table 3 comparing a single-thread CPU implementation to OpenACC, native CUDA, and OCCA (CUDA backend) implementations.

Table 3 Transport kernel wall-clock time (seconds) on NVIDIA V100.

Mesh	CPU Serial	OpenACC	CUDA	OCCA
ex20	8.78790	0.83308	0.02240	0.01299

The initial OpenACC implementation was not only 50x slower than the OCCA implementation (described hereafter) but slower than the serial CPU implementation, highlighting the importance of GPU data layout and thread placement. This example represents a naive OpenACC implementation with minimal source code changes; improved performance can be obtained by taking data layout and memory use into consideration. However, such improvements would require significant source code changes, overriding the original benefits of the directive-based approach.

Since performance is a mandatory prerequisite, the transport kernel was also ported using the native CUDA language. Initial off-loading efforts highlighted the benefits of OCCA for porting source code quickly while achieving satisfactory performance. For example, porting the transport-properties kernel directly with CUDA took approximately seven days, whereas the OCCA version took just one day. Even though both workflows produced CUDA source code, the OCCA version achieved better computational performance, as seen in the above table. The direct CUDA implementation was slightly slower due to the use of read-only texture memory for storing transport parameters (e.g., log tables) versus the OCCA implementation employing shared memory, which has higher bandwidth and very low latency.

Additionally, the OCCA-version of the code provided hardware portability, as kernels were initially developed on an AMD GPU (Radeon VII) and then tested on an NVIDIA GPU (V100). Although high performance is possible using native CUDA, the developer workflow comes with significant overhead and implementation burdens while sacrificing portability. For example, changes in memory environments (e.g., shared memory variables) or thread assignments cause changes to propagate across multiple levels of the source code as the kernel launch mechanisms (e.g., allocating a shared memory buffer passed as a kernel function parameter) require adjustments for every update.

OCCA programming model

To overcome these issues, we adopt the OCCA framework [39], an open-source thread-programming abstraction model, enabling GPUs, CPUs, and FPGAs to achieve performance portability. It allows developers to write performance-portable code in the OCCA Kernel Language (OKL) [40], an intermediate representation for shared-memory parallelism, and is a simple extension of the C programming language. This intermediate representation is just-in-time compiled for various backends, including CUDA, HIP, OpenCL, SYCL, and OpenMP, thus enabling NVIDIA, AMD, and Intel GPUs with one codebase. OKL extends the C language using `@attributes` for code transformations and applies restrictions for exposing loop-based parallelism. Additional attributes and interoperability provided by OCCA enable extensive use of accelerator hardware features. While this approach does require code rewrite in OKL, the kernel language closely resembles the original CPU source code, even in the case of Fortran-to-C conversion. First, unlike hardware-specific languages such as CUDA, the exposure of loop parallelism and different levels of parallel granularity at the block and thread level are annotated explicitly in the OKL kernel.

For example, Listing 1 shows a Fortran 90 subroutine for adding two vectors. Using the CUDA language, the code separates into two components, where the inner computational kernel from a `for/do-loop` structure is maintained in a `__global__` function but with the `for-loop` construct removed (Listing 2), and an explicit CUDA kernel launch function (Listing 3 lines 5-8), where the developer needs to explicitly set the number of threads per block (i.e., `threadsPerBlock`) and calculate the number of thread blocks (i.e., `blockSize`) so that the total number of threads is greater than or equal to the number of vector entries `N`. Additionally, the developer needs to explicitly calculate the thread variable `id` using the CUDA-specific variables `blockDim.x`, `blockIdx.x`, and `threadIdx.x` within the kernel.

```fortran
1  SUBROUTINE add_vectors(N,A,B,C)
2    INTEGER,           INTENT(IN)  :: N
3    DOUBLE PRECISION, INTENT(IN)  :: A(N),B(N)
4    DOUBLE PRECISION, INTENT(OUT):: C(N)
5    IMPLICIT NONE
6    INTEGER :: id ! local variable
7
8    DO id = 1,N
9      C(id) = A(id) + B(id) ! Work to be parallelized
10   ENDDO
11 END SUBROUTINE
```

Listing 1 Fortran 90 subroutine for adding two vectors.

```
__global__
void vectorAdd(const int N, const double* A, const double* B, double* C) {
  const int id = threadIdx.x + blockIdx.x * blockDim.x;
  if (id < N) {
    C[id] = A[id] + B[id];
  }
}
```

Listing 2 CUDA kernel implementation with loop-level parallelism for adding two vectors.

```
#include <cuda_runtime.h>
int main() {
  // ... allocate & initialize host/device memory: N = number of elements
  // Set threads per block and compute number of thread-blocks
  int threadsPerBlock = 256;
  int blockSize = (N + threadsPerBlock - 1) / threadsPerBlock;
  // CUDA kernel launch
  vectorAdd<<<blockSize, threadsPerBlock>>>(N,A,B,C);
}
```

Listing 3 CUDA host pseudo-code with kernel launch mechanism.

```
@kernel
void vectorAdd(const int N, const double* A, const double* B, double* C) {
  for(int id = 0; id < N; ++id; @tile(256, @outer, @inner)) {
    C[id] = A[id] + B[id];
  }
}
```

Listing 4 OCCA Kernel Language (OKL) implementation for adding two vectors.

```
#include <occa.hpp>
int main() {
  // ... allocate & initialize host/device memory: N = number of elements
  // OCCA device setup -- use CUDA backend
  occa::device device("mode: 'CUDA', device_id: 0");
  // Load and build the kernel from the addVectors.okl file
  occa::kernel vectorAdd = device.buildKernel("addVectors.okl", "vectorAdd");
  // OCCA kernel launch
  vectorAdd(N,A,B,C);
}
```

Listing 5 OCCA host pseudo-code with kernel launch mechanism.

Listing 4 shows the equivalent implementation using OCCA and the corresponding host pseudo-code in Listing 5. The `for-loop` construct using OKL is maintained (though converted to C-style language) and decorated with a `@tile` attribute for explicit thread-block and thread-level parallelism injected, thereby closely resembling the original source code. The keyword `@outer` explicitly assigns `thread-block parallelism` (e.g., `blockIdx.x`), while `@inner` explicitly assigns *thread-level parallelism* (e.g., `threadIdx.x`). OCCA **automatically** determines the thread-block dimensions from the `for-loop` parameters. To handle the kernel launch mechanism, OCCA also automatically generates the source code kernel function call, including allocating and passing any shared memory needed for the compute kernel, at run-time. *This allows developers to quickly test various threading strategies for parallelism and memory spaces at run-time using OKL.*

In addition to these thread-parallelism attributes, OKL provides two special memory spaces – `@shared` and `@exclusive` – for leveraging GPU-specific memory types. The `@shared` attribute enables parallel threads/workitems to share data. The `@exclusive` attribute is similar to thread-local storage, where a single variable has one value per thread. For the NVIDIA GH100 GPU, for example, the latency of shared memory is approximately 15x faster* than

*NVIDIA H800 latency clock cycles: shared memory 29.0, global memory 478.8.

global memory [41]. As the OCCA Kernel Language is *just-in-time* (JIT) compiled – with source caching – these features allow developers to leverage GPU devices to high potential with minimal developer effort, particularly for performance optimization workflows.

OKL further enables users to write Fortran-like code containing commonly used multi-indexed arrays with ease using the attribute `@dim`. Listing 6 shows an archetypal example of a three-dimensional array accessed with the `@dim` attribute. Traditionally, code transformation from Fortran to C requires manual arithmetic of linear array indexing. However, as seen in this example, no such action is needed as the `@dim` definition allows seamless compile-time auto-transformation for multi-indexing through the specification of square (linear indexing) or regular (multi-indexing) brackets. This functionality is extensible in the number of indices. Additionally, OKL provides a second optional compile-time auto-transformation attribute `@dimOrder`, which allows the developer to specify the execution order for each index (e.g., row-major for two-index arrays). When `@dimOrder` is absent in the presence of `@dim`, **the left-most index is the fastest**, *mimicking the Fortran language*. For reference, Listing 7 shows the equivalent Fortran 90 subroutine.

```
@kernel
void indexing_function(int nvar,int nelem,double *a){
  double @dim(nvar,nvar,nelem) @dimOrder(0,1,2) *A = a; // optional: @dim
                                                        // optional: @dimOrder
  // local variables
  double a1,a2;

  for(int e = 0; e < nelem; ++e; @outer){
    for(int j = 0; j < nvar; ++j; @inner){
      for(int i = 0; i < nvar; ++i; @inner){
        a1 = A(i,j,e);                    // Fortran-based array indexing
        a2 = a[e*nvar*nvar + j*nvar + i]; // C-based linear indexing
        // ...
    }
  }
}
```

Listing 6 OKL memory access via linear and array indexing. Keywords `@dim` allows multi-index memory access via parenthesis; `@dimOrder` allows assignment of the order of execution for each index.

```
SUBROUTINE indexing_function(nvar, nelem, A)
  INTEGER, INTENT(IN) :: nvar
  INTEGER, INTENT(IN) :: nelem
  REAL,    INTENT(OUT):: A(nvar,nvar,nelem)
  IMPLICIT NONE

  ! local variables
  INTEGER :: i,j,e
  REAL    :: a1

  DO e = 1,nelem
    DO j = 1,nvar
      DO i = 1,nvar
        a1 = A(i,j,e) ! array indexing
      ENDDO
    ENDDO
  ENDDO
END SUBROUTINE
```

Listing 7 Fortran90 memory access via multi-index arrays.

Finally, OCCA provides a simple application programming interface (API) to C, C++, and Fortran for managing device memory, device selection, and execution of OKL kernels. The OCCA API enables users to select the computing device and execution model at run-time (e.g., select NVIDIA GPU with device ID #1 with kernels translated into OpenCL), recalling that the code transpiling process of the OKL kernels also occurs at run-time. After selecting the device and compute mode, developers construct kernel properties that get passed as inputs during the kernel compilation phase. So, a developer has the ability to set various parameters, such as variable types (e.g., `kernelProps["defines/tFloat"] = "double";` or even *function names* (e.g., `kernelProps["defines/riemann_flux_okl"] = "riemann_roe_okl";`). The latter example enables the same capabilities as function pointers in the C-language or procedure pointers in the Fortran 2003 standard. Lastly, OCCA provides a utility command-line interface for debugging, where developers can check the computing environment (e.g., CUDA compiler and flags), the available computing modes and device backends on the system, and direct kernel translation to see the transpiled kernel output for easy debugging (e.g., `occa translate -mode 'CUDA' addVectors.okl`).

B. GPU Acceleration Strategy

The 2D finite-volume real-gas CFD code uses an efficient edge-based data structure for evaluating the inviscid and viscous fluxes. Cell-coloring is employed to avoid data race conditions during the residual evaluation, and a cell-reordering scheme based on the Reverse Cuthill-McKee method [42], in combination with cell renumbering, is used to maximize coalesced memory access. Evaluation of the source term requires the computation of the chemical reactions in the source term in eq. (1). For the 5-species model employed in the CFD solver, 17 chemical reactions are computed for every cell of the computational grid, and efficient acceleration of the evaluation of the source term is critical to the overall performance of the solver.

Since the computation of the chemical reactions in a mesh element is independent of the chemical reactions computed in the other elements, the evaluation of the source term is an ideal candidate for acceleration on modern GPUs that support a Single-Instruction, Multiple-Thread (SIMT) execution model as will be shown in the results. The source term is evaluated using a for-loop over all the cells in the CFD grid, and OCCA `@attributes` distributes the grid over the many threads of the many-core accelerator, blocking 32 elements per GPU thread.

The same multi-threading strategy is used to accelerate the computation of the transport coefficients. Finally, offloading the existing line-implicit solver to the many-core accelerator enables the execution of all the computationally intensive operations on the GPU, essentially realizing the GPU acceleration of the entire CFD solver. At the beginning of the simulation, the CPU reads the computational mesh from files and initializes the memory arrays needed for the numerical solution of the flow field. These arrays are placed in GPU memory, where they reside for the duration of the simulation. At the end of the simulation, the solution array is transferred back to the CPU memory and written to a file for post-processing purposes.

We also investigate the implementation and performance of alternate solution strategies for cases with many species, including reducing solver memory requirements for overcoming limited memory on GPU architectures. To this end, we considered a Newton-Krylov method with different preconditioning techniques, focusing on the performance of reduced-precision preconditioners. Here, the linear system arising from the quasi-Newton method is solved using single-precision Jacobian blocks.

IV. Results

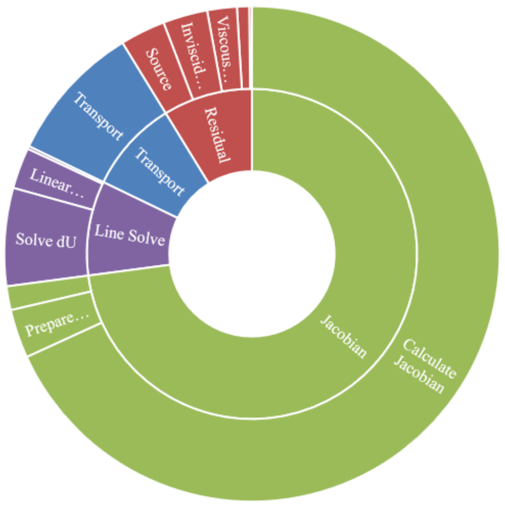
A. Testing Platform

The platform used to benchmark the CPU and GPU implementations is an NVIDIA DGX-1 system containing dual-socket 20-core Intel Xeon E5-2698v4 CPUs at 2.20 GHz and eight NVIDIA Tesla V100 SXM2 32GB GPUs. For OpenMP profiling, we utilize one OMP thread per core, resulting in a total of 40 OMP threads across two CPUs. For OpenACC and OCCA profiling, we utilize one 32GB V100 GPU, which has a memory bandwidth of 900 GB/s with FP32 15.7 TFLOP/s and FP64 7.8 TFLOP/s peak performance. Additionally, we provide benchmarks using an AMD Radeon VII GPU containing 60 compute units, 16GB HBM2 memory with up to 1024 GB/s bandwidth, and FP32 13.8 TFLOP/s and FP64 3.46 TFLOP/s peak performance.

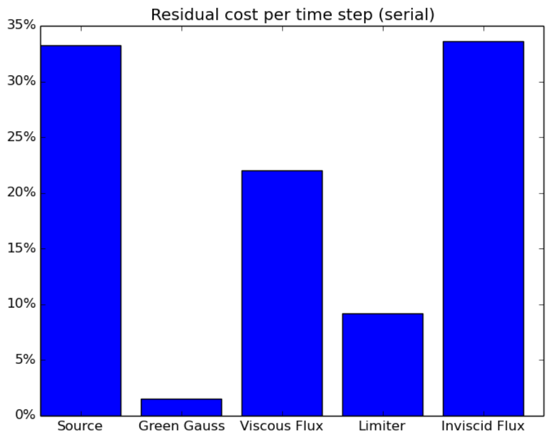
B. CPU Results: Application Profile

The CPU computational cost of one nonlinear iteration of the finite-volume solver is shown in Fig. 3. Figure 3(a) provides a proportional representation of the primary kernel computational code breakdown of one nonlinear iteration of the solver using five linear iterations of the line-implicit Jacobi solver (annotated Line Solve). As demonstrated in Fig. 3(a), the construction of the reduced-order Jacobian dominates the cost of one nonlinear iteration of the CFD solver. We note this relative cost can be highly variable, depending on the number of linear solver iterations performed within each nonlinear cycle, during which we freeze the Jacobian. However, the current scenario corresponds to the early stages of convergence, when the pseudo-time step is small, significantly increasing the solve time. Beyond the Jacobian construction, the other principal kernels include the evaluation of the transport coefficients, the residual vector, and the execution of the linear line solver.

A breakdown of the CPU computational cost of one residual evaluation for the 5-species gas mixture with 17 chemical reactions is shown in Fig. 3(b). Even for this limited number of species, computation of the source term is the most expensive operation during a residual evaluation. Furthermore, since the number of chemical reactions grows superlinearly with the number of species in the gas mixture [43], the computation of the finite-rate chemistry in the source term can dominate the cost of a residual evaluation. To demonstrate efficient execution of the entire real-gas hypersonic CFD code on GPU architectures, each one of these principal kernels must be ported and optimized for efficient execution on GPUs, and the code must be executed with all memory residing on the GPU card, with transfer to the CPU host only occurring at the end of the solution process for output to disk.



(a) CPU computational cost breakdown for the four primary kernels: Transport, Residual, Jacobian, Line Solve. Sub-kernel breakdown is provided for the Residual and Line Solve.



(b) CPU computational cost percentage breakdown of one residual kernel evaluation. Source and Inviscid Flux kernels dominate the overall cost of a residual evaluation.

Fig. 3 CPU computational cost breakdown over kernels for one nonlinear iteration of the solver.

C. GPU Results: Transport Properties Kernel

Here we focus on detailed evaluation of the performance of two of the important kernels of the real-gas CFD code, namely the evaluation of the transport coefficients in this subsection, and the execution of the linear solver in the following subsection. At each nonlinear update, local values of the transport properties, such as viscosity, thermal conductivity, and diffusion coefficients must be calculated. To compute these properties, the solver uses a transport model based on collision integrals. The integrals account for the interaction of the individual species at the molecular level. These transport properties are computed by linearly interpolating the tabulated collision integrals at two temperatures for the given set of species. For the 5 species considered for this work, a total of 60 collision integrals are given for two temperatures at 2000 K and 4000 K. More details can found in the original work on the solver [19].

The transport quantities are calculated using face-centered quantities and are utilized for the viscous flux computation in the residual kernel. The face primitive state is obtained by averaging the left and the right cell primitive state values. These average face state values are used to compute the transport quantities. The computation of all faces for a given face group (out of four total in the present case) can be performed in parallel, and each thread is assigned to a face to compute the transport quantities. The major aspect of this kernel computation involves collision integral table look-up for a given temperature. Given the read-only operation, the kernel makes use of shared device memory for these values. This kernel also uses shared device memory for several constant parameters, such as free-stream values and log tables. In addition, thread divergence has been minimized by eliminating `if/else` operations. The Transport kernel is decomposed over the GPU by blocking 32 faces together within a color group mapped to a thread-block with each color group executed in parallel across thread-blocks. Roofline analysis reveals the kernel is compute bound for the FP64/double precision arithmetic reaching 2.33 TFLOP/s compute performance on the **ex06** mesh.

D. GPU Results: Linear Solver Kernels

Acceleration of the solution of the linear system in eq. (3) is enabled by the OCCA implementation of multiple linear solvers and a Newton-Krylov preconditioned GMRES solver. We describe the approaches taken herein and provide performance evaluations of the GPU implementations in the section immediately following. The linear solvers and preconditioners used in this project are divided into point/line solvers and are cast as linear solvers. Their general form is found by placing the linearized equation resulting from Newton's method in eq. (4).

$$\begin{aligned} \left[A\right] x &= b \\ \left[A\right] \left(x^{k+1} - x^{k}\right) &= b - \left[A\right] x^{k} \\ \left[P\right] \delta x^{k} &= b - \left[A\right] x^{k} = -r^{k} \end{aligned} \tag{4}$$

The matrix $\left[P\right]$ denotes the preconditioner matrix or the approximate matrix that must be inverted, δx^k represents the $k^{\texttt{th}}$-iteration solution update, and r^k is the residual of the linear system. When this system is derived from a Newton linearization, b corresponds to the nonlinear residual vector, and $\left[A\right]$ denotes the Jacobian matrix.

Point-Implicit Jacobi (PIJ)

The first linear solver/preconditioner corresponds to a point-implicit Jacobi solver, whereby the block diagonal elements of the reduced-order Jacobian are inverted to determine the solution update. When this solver is applied multiple times to the linear system within the Newton iteration, it is referred to as a Jacobi linear solver. When it is applied only once before moving to the next non-linear iteration, it is often referred to as a non-linear Jacobi solver. The Jacobi iteration is given as follows:

$$[D]\delta x^k = -r^k \tag{5}$$

The matrix D consists of the diagonal blocks of the $\left[A\right]$ matrix, which are inverted to yield the solution update. The block size varies based on the number of species and is inverted using LU factorization. For the non-linear Jacobi solver, the right-hand-side of eq. (5) is the non-linear residual, while the linear Jacobi solver requires the linear residual on the right-hand side, given as:

$$r^k = [A]x^k - b \tag{6}$$

Table 4 reports the timing statistics for the linear Jacobi solver on all meshes for CPU and GPU implementations per nonlinear iteration using five linear iterations. Note that the GPU metrics provide three versions of the solver: **FP64**, **Mixed**, and **FP32**. The **FP64** implementation uses `double` precision floating point values for all Jacobian blocks and the linear residual as shown in eq. (6). The **Mixed** implementation down-converts the Jacobian blocks to `single` precision floating point for all Jacobian blocks, but maintains double precision for the linear residual vector.

Following reference [44], the Jacobian entries are first computed via the Newton linearization in double precision and the block diagonals are LU factored. The factored diagonals, along with the original block entries of the $\left[A\right]$ Jacobian matrix are then down-converted to single precision. The first term in the linear residual (i.e., $\left[A\right] x$) is computed in single precision using the lower precision entries for the $\left[A\right]$ matrix. This vector is then upgraded to double precision and subtracted from the double precision nonlinear residual vector (i.e., the b vector) which is held constant throughout the nonlinear cycle. As shown in [44], this approach reproduces nearly identical convergence histories as the double precision linear solver.

Lastly, the **FP32** implementation down-converts both the Jacobian blocks and the linear residual vector. However, the convergence properties of the linear solver in this case are altered and convergence stalls at residual norms of approximately 10^{-6}. As can be seen from the tabulated results, the mixed precision version of the linear solver achieves over 10% to 15% speedup compared to the double precision version, while the single precision version achieves over 60% speedup. Although the latter has the undesirable effects of convergence stalling, it may possibly be used for cases where low convergence tolerances are acceptable, such as in the initial stages of convergence. Except on the coarsest mesh, the double precision solver runs more than 500 times faster on the V100 GPU compared to a single CPU core.

Table 4 Point-Implicit Jacobi (PIJ) solver timing performance (seconds) comparisons between CPU and GPU per nonlinear solve iteration. The five-species system generates 9x9 Jacobian matrix blocks for each mesh element. Five linear iterations are employed within a single nonlinear iteration.

PIJ: 5 iterations	**CPU**	**GPU: NVIDIA V100**			**Serial vs.**
Mesh	**Serial**	**FP64**	**Mixed**	**FP32**	**V100 FP64**
ex05	0.05936	0.000233	0.000216	0.000179	254.8x
ex06	2.22167	0.004398	0.003909	0.002739	505.2x
ex10	6.17133	0.011997	0.010645	0.007438	514.4x
ex20	25.2955	0.047616	0.042322	0.029524	531.2x

Line-Implicit Jacobi (LIJ)

In order to overcome the stiffness caused by the anisotropic grid stretching encountered in boundary layer flows, a line relaxation algorithm is employed. For this algorithm, lines are constructed across the boundary layer and are treated as the fundamental units of the linear solver. In this way, the Jacobian matrix is divided into blocks corresponding to each line, uncoupled from neighboring lines. With the basic blocks of the solver identified, the Jacobi approach is used to invert the global matrix once it has been partitioned into line blocks. The Jacobi line solver is represented as:

$$[M]\delta x^k = -r^k \tag{7}$$

where $[M]$ is a block tridiagonal matrix. Each $[M]$ is constructed by connecting the diagonal block element with the element's two off-diagonal block elements. Since each line is decoupled, trivial parallelization over the lines is obtained.

Figure 4 illustrates an example of three vertical lines with full Jacobian matrix shown in eq. (8), where the colors indicate line affiliation, with eq. (9) displaying the tridiagonal $[M]$ matrix for Line 2.

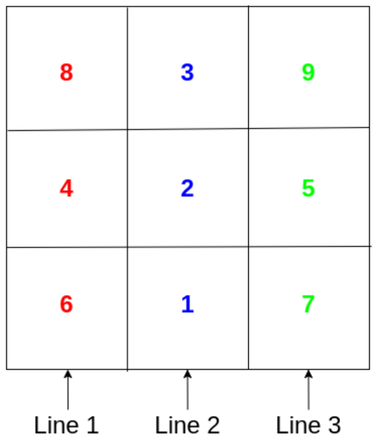


Fig. 4 Sample mesh containing 3x3 elements with three implied lines constructed vertically. Each line defines a line-implicit stencil with same-colored elements being tightly coupled and solved directly using a block-variable Thomas algorithm for tridiagonal matrices.

$$[J] = \begin{bmatrix}
\left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_1}\right] & \left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_2}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_6}\right] & \left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_7}\right] & \mathbf{0} & \mathbf{0} \\
\left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_1}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_2}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_3}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_4}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_5}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_2}\right] & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_3}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_8}\right] & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_9}\right] \\
\mathbf{0} & \left[\frac{\partial \mathbf{R}_4}{\partial \mathbf{Q}_2}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_4}{\partial \mathbf{Q}_4}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_4}{\partial \mathbf{Q}_6}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_4}{\partial \mathbf{Q}_8}\right] & \mathbf{0} \\
\mathbf{0} & \left[\frac{\partial \mathbf{R}_5}{\partial \mathbf{Q}_2}\right] & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_5}{\partial \mathbf{Q}_5}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_5}{\partial \mathbf{Q}_7}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_5}{\partial \mathbf{Q}_9}\right] \\
\left[\frac{\partial \mathbf{R}_6}{\partial \mathbf{Q}_1}\right] & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_6}{\partial \mathbf{Q}_4}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_6}{\partial \mathbf{Q}_6}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\left[\frac{\partial \mathbf{R}_7}{\partial \mathbf{Q}_1}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_7}{\partial \mathbf{Q}_5}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_7}{\partial \mathbf{Q}_7}\right] & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_8}{\partial \mathbf{Q}_3}\right] & \left[\frac{\partial \mathbf{R}_8}{\partial \mathbf{Q}_4}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_8}{\partial \mathbf{Q}_8}\right] & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_9}{\partial \mathbf{Q}_3}\right] & \mathbf{0} & \left[\frac{\partial \mathbf{R}_9}{\partial \mathbf{Q}_5}\right] & \mathbf{0} & \mathbf{0} & \mathbf{0} & \left[\frac{\partial \mathbf{R}_9}{\partial \mathbf{Q}_9}\right]
\end{bmatrix} \tag{8}$$

$$[M]_{\text{Line 2}} = \begin{bmatrix}
\left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_1}\right] & \left[\frac{\partial \mathbf{R}_1}{\partial \mathbf{Q}_2}\right] & \mathbf{0} \\
\left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_1}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_2}\right] & \left[\frac{\partial \mathbf{R}_2}{\partial \mathbf{Q}_3}\right] \\
\mathbf{0} & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_2}\right] & \left[\frac{\partial \mathbf{R}_3}{\partial \mathbf{Q}_3}\right]
\end{bmatrix} \tag{9}$$

Using this matrix splitting, the block matrix system is solved using the block Thomas algorithm for tridiagonal matrix systems. Consider the following block tridiagonal matrix:

$$\big[[A_i], [B_i], [C_i]\big]\, x_i = r_i, \;\; \forall i = 1, N \tag{10}$$

where $[A_i]$ is the lower-diagonal entry (i,i-1), $[B_i]$ is the diagonal entry (i, i), and $[C_i]$ is the upper-diagonal entry (i,i+1) in the i^{th} row of the matrix. The block Thomas factorization algorithm is given by Algorithm 1. For a given number of LIJ iterations, the factorization is performed only once and reused for several iterations. Figure 5 illustrates the linear convergence of the point- (Jacobi) and line-implicit solvers for one nonlinear step of the reduced-order Jacobian system with pseudo-time stabilization using a CFL of 0.1 on the **ex06** mesh. Even at these low pseudo-time step sizes, characteristic of early phase of the convergence history, the benefits of the line-implicit solver are clearly demonstrated as the convergence rate is well maintained to machine double-precision whereas the point-implicit Jacobi solver suffers severe slowdown induced by the stiffness from the boundary layer.

Algorithm 1 Block Thomas Factorization

```
$[B'_1] = LU([B_1])$
for $j = 2, N$ do
    $[B'_j] = LU([B_j] - [A_j][B'_{j-1}]^{-1}[C_{j-1}])$
end for
```

Algorithm 2 Block Thomas Iteration

```
for $k = 1, N_{iteration}$ do
    $y_1 = [B'_1]^{-1} r_1$                                ▷ Forward Substitution
    for $j = 2, N$ do
        $y_j = [B'_j]^{-1}(r_j - [A_j] y_{j-1})$
    end for
    $x_N = y_N$                                            ▷ Back Substitution
    for $j = N - 1, 1$ do
        $x_j = y_j - ([B'_j]^{-1}[C_j]) x_{j+1}$
    end for
    $r_j = -R_j - [J] x_j$                                 ▷ Linear Residual
end for
```

For GPU-acceleration of the line solver kernel, we assign coarse-grain thread-block parallelism across the lines in the mesh, as their sparse linear systems are independent of each other. Within each line, a tridiagonal linear system is solved using the block-Thomas algorithm. This algorithm is inherently sequential as one forward and back substitution pass is needed to complete the solve, thus direct coarse-grain parallelism within each line is prohibited. However, fine-grain parallelism is injected into the block matrix-vector products, such as $[A_j]y_{j-1}$ and $[B'_j]^{-1}[C_j]x_{j+1}$, where each row of the matrix is assigned to a thread. Shared memory is utilized throughout the kernel to store and share the matrix blocks and intermediate vector values across all threads involved in the computation.

The roofline profile for the δU-solve kernel – the forward and backward solve of the point- or line-implicit solver – shown in Fig. 6 highlights near-optimal bandwidth throughput indicated by the red dot (FP64 performance) lying close to the blue roofline curve, which represents the maximum achievable performance for a specific arithmetic intensity.

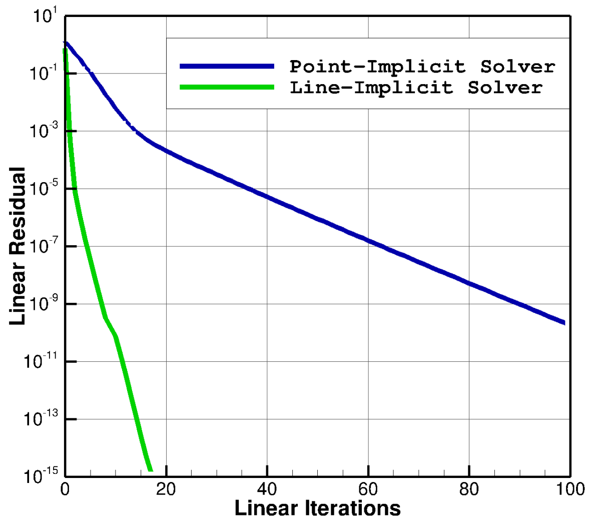


Fig. 5 Linear solver convergence comparison between point- and line-implicit solvers for nonlinear residual iteration with small pseudo-time step.

The linear residual calculation at the end of the Block Thomas Iteration in Algorithm 2 is separated into an isolated kernel. The linear residual kernel is composed of a sparse matrix-vector product and a resultant vector-vector addition operation. We are able to fully coarse-grain parallelize these operations to obtain nearly optimal bandwidth as shown Fig. 7. Table 5 reports the performance statistics for the Line solver for CPU and GPU implementations. The results depict the timings for 5 linear line-solve iterations, which can be seen to require approximately 30% more time on a single CPU core than the equivalent point-implicit solver in Table 4. Here again, timings are given for the full double precision, mixed precision, and single precision version of the line solver. Overall, the double precision line solver achieves a speedup of approximately 430x over a single CPU core on the two finest grid levels for this case.

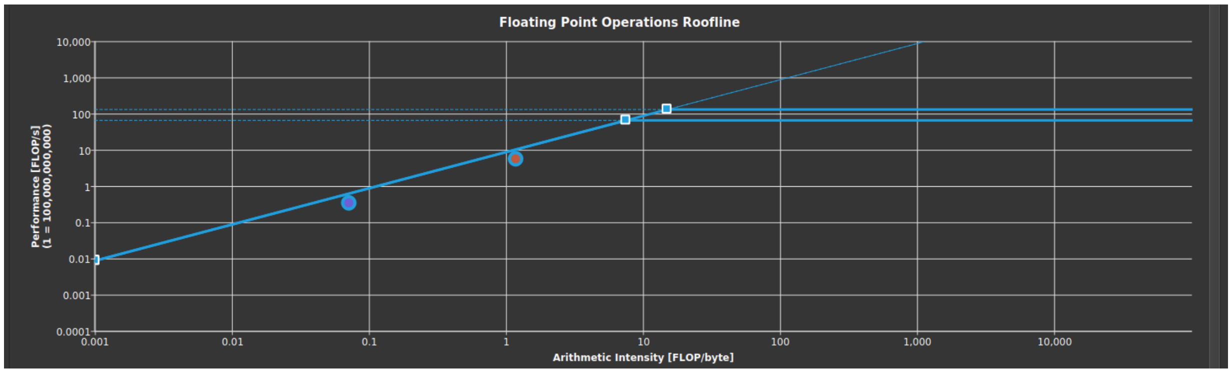


Fig. 6 Line solver δU-solve kernel roofline analysis on a 32GB NVIDIA V100-SXM2 GPU. (The red dot indicates FP64 performance and the violet dot indicates FP32 performance which is inconsequential for this kernel).

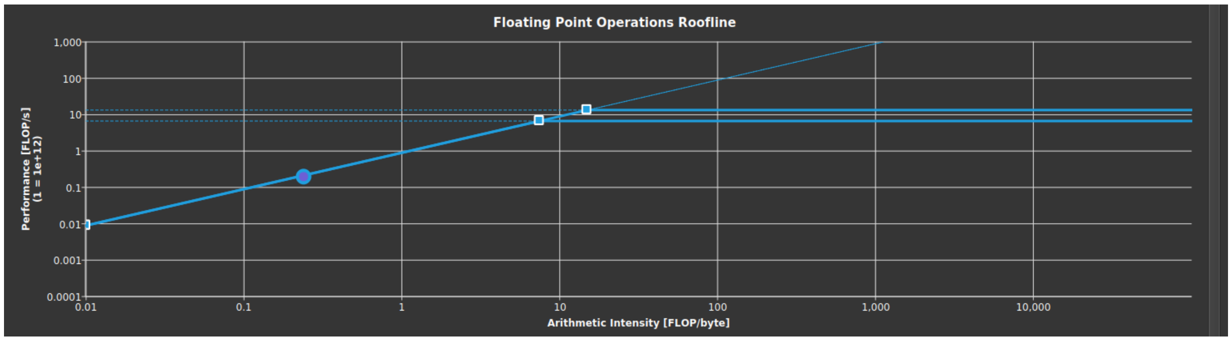
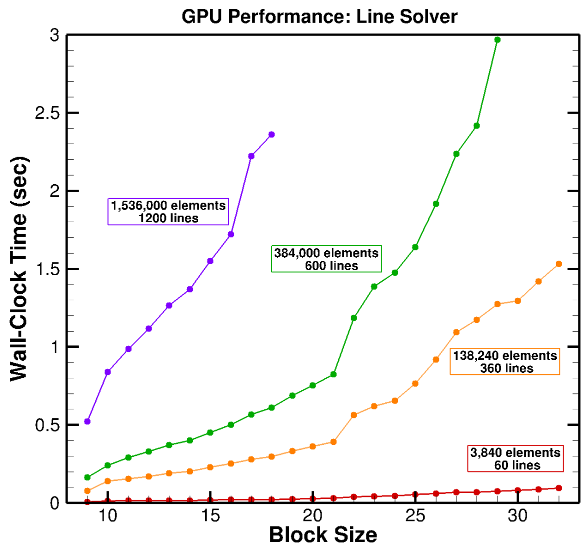


Fig. 7 Line solver linear residual kernel roofline analysis on a 32GB NVIDIA V100-SXM2 GPU. The violet dot indicates FP64 performance.

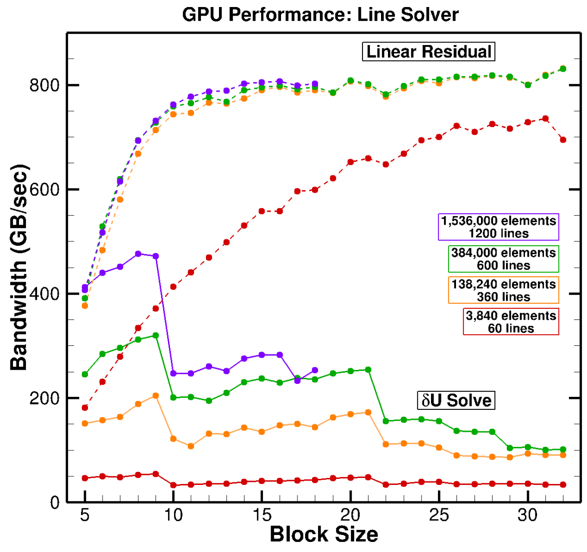
Table 5 Line-implicit solver timing performance (seconds) comparisons between CPU and GPU per nonlinear solve iteration. The five-species system generates 9x9 Jacobian matrix blocks for each mesh element and a total of five linear iterations are employed.

LIJ: 5 iterations	**CPU**	**GPU: NVIDIA V100**			**Serial vs.**
Mesh	**Serial**	**FP64**	**Mixed**	**FP32**	**V100 FP64**
ex05	0.07654	0.000943	0.001116	0.000976	81.2x
ex06	2.77703	0.008470	0.009204	0.009205	327.9x
ex10	8.24317	0.018741	0.019479	0.015401	439.8x
ex20	33.1932	0.076733	0.060517	0.046786	432.6x

Lastly, we have performed a study varying the block sizes of the Jacobian blocks, corresponding to different numbers of considered chemical species. Figure 8 illustrates the wall-clock times and their corresponding bandwidth numbers for the sub-kernels in the line smoother kernel. For the block size of 9 executed on the largest mesh containing 1.536 million elements, the δU-solve kernel achieves 472 GB/s bandwidth and the linear residual kernel achieves 731 GB/s. In the left figure, two linear trends are pronounced for the 138,000 and 384,000 element meshes with rapid change occurring at a block size of 22. This rapid change is reflected in the right figure of the bandwidth performance curves corresponding to the δU-solve kernel. This performance drop is caused by register contention across the V100 GPU, causing a reduction of thread-blocks allowed to be executed in parallel. This is a known as the Tail Effect [45] and can be mitigated by leveraging the `__launch_bounds__` attribute in CUDA by requesting a minimum number of thread-blocks to execute in parallel, thereby limiting the number of registers allowed per thread-block. In addition to the performance drop at a block size of 22 for all meshes, we also see a similar drop at a block size of 10, which is again as the Tail Effect, confirmed using the NVIDIA Nsight Compute tool. An additional approach to addressing this issue is to batch lines together into a single thread-block, again, easing the register contention issue; the current implementation assigns one thread-block per line. The five species model used in the 2D flow solver requires a block size of 9, which performed well with the current implementation.



(a) Wall-clock time (sec) to solve the linear system of equations for each mesh.



(b) Bandwidth (GB/s) performance: (dash-dot) linear residual, (line-dot) δU-solve.

Fig. 8 Line-implicit solver FP64 performance for varying block sizes, i.e., number of chemical species, on a 32GB NVIDIA V100-SXM2 GPU. Each color represents a mesh: (red) `ex05`, (orange) `ex06`, (green) `ex10`, (purple) `ex20`.

Preconditioned GMRES(m)

In addition to point- and line-implicit solvers, the investigation of Newton-Krylov approaches preconditioned using the GPU-accelerated point- and line-implicit solvers discussed previously is conducted. We implement the k-step version of the Generalized Minimal Residual method (GMRES(m)) which fixes the Krylov subspace dimension to at most m vectors. If the convergence tolerance is not achieved with all subspace vectors, the solution is updated using the linear combination of the vectors and the subspace search is restarted. Though this approach may limit the convergence rate, the fixed memory allocation strategy is essential as GPUs have relatively limited memory capacity. Although we have implemented mixed and single precision versions of the preconditioner, here we provide the performance statistics for the line-preconditioned GMRES solver using the double-precision variant of the line smoother. Results are tabulated in Table 6. Each nonlinear cycle consists of two Krylov vectors in the GMRES linear solver, using a single sweep of the line solver as a preconditioner. The use of a small number of preconditioning steps and Krylov vectors enables the assessment of the efficiency of the additional steps required in the Newton-Krylov solver compared to the preconditioner itself or the linear line solver approach, since for large numbers of preconditioning sweeps, the timings can be expected to be dominated by the line-solver phase. Overall, with these settings, the Newton-Krylov solver exhibits somewhat higher speedups on each mesh level compared to the line solver alone, topping out at 595x on the finest mesh compared to a single CPU core. These results point to the efficient GPU execution of the additional Jacobian-vector products and other vector product operations required within the Newton Krylov solver. Future investigations should look at the effect of single precision preconditioner implementation within the Newton-Krylov solver, where the effects of degraded convergence may be less important in the preconditioning stage, as the GMRES solver remains in full double precision.

Table 6 Line-preconditioned GMRES solver performance (seconds) comparisons between CPU and GPU per nonlinear solve iteration. The five-species system generates 9x9 Jacobian matrix blocks for each mesh element. One iteration of the FP64-variant line-implicit smoother is used for preconditioning and two Krylov vectors are used within the GMRES solver

Mesh	Serial	GPU: V100	Serial vs. V100
ex05	0.1230	0.001149	232.6x
ex06	3.8528	0.010079	382.3x
ex10	11.3768	0.023914	475.7x
ex20	45.6677	0.076719	595.3x

E. GPU Results: Application Profile

The entire multi-species real gas 2D Navier-Stokes hypersonic code has been fully ported to advanced many-core accelerators including NVIDIA V100 and AMD Radeon VII GPUs, using the same source code within the OCCA framework, based on the porting of the individual kernels described in the previous sections. Herein, we summarize the dominant flow solver kernels and their performance profiles across the four meshes, and compare these results to serial and OpenMP (OMP) CPU implementations running on a single compute node, and partial OpenACC GPU implementations. The four subtables in Table 7 show the primary kernel (Transport, Residual, Jacobian, and Line Solve) performance measurements on each mesh level. The CPU results provided here are executed on a dual-socket node composed of Intel Xeon E5-E5-2698v4 CPUs. The OMP results used 40 CPU threads invoked through loop-level parallelism, whereas the GPU results (OpenACC and OCCA) used one 32GB V100 GPU or (OCCA) one 16GB AMD Radeon VII GPU. The OCCA implementation utilized just-in-time compilation (with source caching) to the CUDA backend for the NVIDIA GPU and the HIP backend for the AMD GPU. As mentioned previously, the OpenACC results are incomplete, as this approach was abandoned early on, after it became obvious that suitable performance with this approach would be difficult to achieve in the absence of considerable source code modifications. The OCCA results, on both NVIDIA and AMD GPUs, consistently outperformed the OpenMP implementation by about one order of magnitude. However, it must be emphasized that the OpenMP implementation may not be optimal, which is why we also choose to compare performance versus a single CPU core.

Table 8 provides the kernel comparisons between the serial CPU and OCCA NVIDIA V100 implementation with listed speedup metrics on the largest mesh **ex20**. Here the largest speedup over serial single core timing is observed for the transport kernel, which is 680x faster on a single V100 GPU versus a single CPU core. On the other hand, the Jacobian kernel achieves the lowest overall speedup at 182x on this mesh. The entire application code achieves a speedup of 207x on a single V100 GPU compared to execution on a single CPU core. Comparing the speedup of the GPU implementation on a per node basis is complicated by the fact that the OpenMP implementation used for intranode parallelism on the dual socket CPUs may not be optimal. Clearly, from the results in Table 7, speedups within a single compute node over the OpenMP implementation of the order of 10x are seen. In Table 9 we provide another metric for assessing potential speedup over an optimal distributed memory CPU-based implementation. The estimates provided are based a conservative lower bound (shown in yellow) and a moderate upper bound (shown in green). The conservative lower bound is estimated by taking the serial CPU wall-clock time and assuming perfect parallel scalability on 40 CPU cores (one dual-socket node). The upper bound estimate is garnered via an approximate weak-scaling test of the serial CPU implementation for each mesh. The approximate weak-scaling test leveraged MPI to launch and time 40 parallel instances of the serial implementation, demonstrating roughly a 30% decrease in performance compared to the perfect scaling assumption. The results show that for larger meshes that saturate the GPU, one NVIDIA 32GB V100 GPU is approximately 5-6x faster than one DGX-1 node containing dual-socket 20-core Intel Xeon E5-2698v4 CPUs for the 2D solver.

Table 7 Flow solver kernel timings (seconds) comparisons for one kernel execution using FP64.
CPU: Dual-socket 2.2GHz 20-core Intel Xeon E5-2698v4 — Peak Bandwidth: 76.8 GB/s
GPU-A: NVIDIA V100 SXM2 32GB — Peak Bandwidth: 900 GB/s
GPU-B: AMD Radeon VII 16GB — Peak Bandwidth: 1,024 GB/s
- OpenMP (OMP) executed with 40 CPU hardware threads
- OpenACC and OCCA executed on single GPU

(* optimization incomplete, - port incomplete)

Mesh: ex05

Time (sec)	Intel Xeon CPU		NVIDIA GPU		AMD GPU
Kernel	**Serial**	**OMP(40)**	**OpenACC**	**OCCA**	**OCCA**
Transport	0.02235	0.00145	0.02829	0.000525	0.002616
Residual	0.02116	0.00243	0.0741*	0.000501	0.001976
Jacobian	0.17100	0.00947	-	0.002670	0.010470
Line Solve	0.01219	0.00207	-	0.000563	0.001752
Total	0.22670	0.01542	-	0.004259	0.016814

Mesh: ex06

Time (sec)	Intel Xeon CPU		NVIDIA GPU		AMD GPU
Kernel	**Serial**	**OMP(40)**	**OpenACC**	**OCCA**	**OCCA**
Transport	0.79398	0.03307	0.12111	0.001488	0.005811
Residual	0.74773	0.03833	0.9972*	0.003194	0.009398
Jacobian	6.40833	0.36511	-	0.037487	0.048907
Line Solve	0.63359	0.05725	-	0.004372	0.013738
Total	8.58363	0.49376	-	0.046541	0.077854

Mesh: ex10

Time (sec)	Intel Xeon CPU		NVIDIA GPU		AMD GPU
Kernel	**Serial**	**OMP(40)**	**OpenACC**	**OCCA**	**OCCA**
Transport	2.23275	0.09292	0.24334	0.003356	0.012078
Residual	2.19236	0.12160	2.6836*	0.007846	0.023307
Jacobian	18.7748	1.01846	-	0.099632	0.111150
Line Solve	1.60053	0.17438	-	0.009330	0.027712
Total	24.8004	1.40736	-	0.120164	0.174247

Mesh: ex20

Time (sec)	Intel Xeon CPU		NVIDIA GPU		AMD GPU
Kernel	**Serial**	**OMP(40)**	**OpenACC**	**OCCA**	**OCCA**
Transport	8.83793	0.31520	0.83308	0.012989	0.039574
Residual	8.49231	0.40948	10.549*	0.029654	0.085749
Jacobian	69.7675	3.36186	-	0.383551	0.396698
Line Solve	8.96476	0.61790	-	0.038530	0.121420
Total	96.0625	4.70444	-	0.464724	0.643441

Table 8 Flow solver kernel wall-clock time (seconds) and speedup comparison on Mesh ex20.
CPU: dual-socket 2.2GHz 20-core Intel Xeon E5-2698v4, Peak Bandwidth: 76.8 GB/s
GPU: NVIDIA 32GB V100 SXM2, Peak Bandwidth: 900 GB/s

Time (sec)	**Serial**	**OCCA**	**Speedup**
Transport	8.83793	0.012989	680.4x
Residual	8.49231	0.029654	286.4x
Jacobian	69.7675	0.383551	181.9x
Line Solve	8.96476	0.038530	232.7x
Total	96.0625	0.464724	206.7x

Table 9 Flow solver total wall-clock time (seconds) and estimated speedup versus 40 CPU cores with MPI.
CPU: dual-socket 2.2GHz 20-core Intel Xeon E5-2698v4, Peak Bandwidth: 76.8 GB/s
GPU: NVIDIA 32GB V100 SXM2, Peak Bandwidth: 900 GB/s

				Speedup Estimate
Mesh	**Elements**	**Serial**	**OCCA**	**MPI(40) vs. OCCA**
ex05	3840	0.22670	0.004259	1.3 - 1.9 x
ex06	138240	8.58363	0.046541	4.6 - 6.3 x
ex10	384000	24.8004	0.120164	5.2 - 6.8 x
ex20	1536000	96.0625	0.464724	5.2 - 6.6 x

`Lower` bound estimate based on ideal scaling of serial code.
`Upper` bound estimate based on approximate weak scaling of serial code.

F. GPU Acceleration: Adjoint Solver

Lastly, we investigate the performance of the adjoint operator in the 2D code on GPU architectures. Since the adjoint operator corresponds to the transpose of the linearized analysis routines, the flow of information is reversed in all adjoint operations, which may impact the required data layouts for optimal GPU performance. The adjoint operation which involves linearized residual-vector products can be thought of as the transpose of the Jacobian-vector products which occur in the Newton-Krylov method. Therefore, by performing the GPU implementation of this important adjoint kernel, we obtain information of the requirements and performance characteristics of an important adjoint component and contrast this with the performance of the forward or analysis mode.

The adjoint vector Λ corresponding to a scalar simulation objective L is obtained as the solution to the following adjoint equation:

$$\left[\frac{\partial R}{\partial Q}\right]^T \Lambda = -\left[\frac{\partial L}{\partial Q}\right]^T \tag{11}$$

Here the matrix on the left-hand side corresponds to the full Jacobian matrix obtained from an exact linearization of the spatial residual of the discretized real gas flow equations. This matrix is considered too large to assemble and invert directly, therefore the adjoint equation is solved using a defect-correction approach following:

$$\left[\frac{\partial R}{\partial Q}\right]^T_{1st} \Delta\Lambda^k = -\left[\frac{\partial L}{\partial Q}\right]^T - \left[\frac{\partial R}{\partial Q}\right]^T \Lambda^k \tag{12}$$

$$\Lambda^{k+1} = \Lambda^k + \Delta\Lambda^k \tag{13}$$

Here the left-hand side (LHS) matrix corresponds to a simplified Jacobian resulting from the linearization of a first-order accurate discretization. The LHS matrix corresponds to the transpose of the same simplified Jacobian used for the analysis problem. The right-hand side (RHS) terms correspond to the (linear) residual of the adjoint problem. The first term on the RHS corresponds to the partial derivative of the scalar objective function with respect to the flow variables. This vector is constant during the entire solution process and is computed on the CPU for expediency and not included in the overall timings. The second term corresponds to a rather complicated matrix-vector product that is at the core of the adjoint evaluation. Equation 12 is solved with 3 passes of the line solver, and the outer iterations (index k) are repeated until the system converges. In the current benchmarking exercise we use 10 outer iterations which requires a new evaluation of the RHS terms (adjoint equation residual) at each outer iteration.

We perform computational tests of the solution of the adjoint problem using a serial CPU core and a single GPU. Table 10 outlines the wall-clock times of the adjoint operator kernels tested on the hypersonic benchmark case meshes using a single CPU core of an Intel Xeon E5-2698v4 and one NVIDIA 32GB V100 SMX2 GPU. For each evaluation, timing of the LHS includes the computation of the Jacobian blocks, transpose operations, and factorization. The timing of the RHS vector includes computation of the second term and addition of the first term in eq.(12). Lastly, the linear system is solved iteratively using three passes of the line-implicit Jacobi solver and ten outer defect-correction iterations. Timings for all kernels are provided along with their cumulative totals. Speedup comparisons are provided for all kernels in the last tabulation in Table 10 showing up to a total speedup factor of 640x on the mesh **ex20**. As seen from the results, the wall-clock time becomes dominated by the computation of the RHS vector, which requires reverse differentiation of the residual, transport, and primitive kernels. Significant performance is achieved using the GPU platform for the RHS kernel, demonstrating an upwards of 819x speedup over the serial CPU implementation. Naturally, the precise speedups obtained for the solution of the adjoint problem will depend on the relative number of inner and outer iterations performed in the defect-correction solver, although we expect equal or better performance compared to the analysis problem due to the efficient execution of the RHS kernel on the GPU device.

Table 10 Adjoint Solver assembly and solve wall-clock time (seconds). Ten outer iterations are performed on the Adjoint system using a frozen left-hand-side (LHS) adjoint system. Assembly of the Adjoint right-hand-side (RHS) is performed for each outer iteration, and three inner iterations of the line-implicit smoother are applied to the linear system. Speedup characteristics between the serial CPU and GPU are provided in the last table.

Intel Xeon E5-2698v (serial)

Time (sec)	**LHS(`1x`)**	**RHS(`10x`)**	**Solve(`10x`)**	**Update(`10x`)**	**Total**
ex05	0.390383	5.210162	0.173891	0.000589	5.775025
ex06	13.93848	189.0698	6.830422	0.028704	209.8674
ex10	38.65058	509.6371	19.12939	0.059581	567.4767
ex20	148.4270	2029.730	91.58548	0.437405	2270.179

NVIDIA 32GB V100 SMX2

Time (sec)	**LHS(`1x`)**	**RHS(`10x`)**	**Solve(`10x`)**	**Update(`10x`)**	**Total**
ex05	0.009977	0.057925	0.009938	0.000102	0.077942
ex06	0.061396	0.278766	0.071435	0.000462	0.412058
ex10	0.144553	0.661647	0.138830	0.001111	0.946117
ex20	0.507954	2.478706	0.552645	0.004164	3.543468

Speedup: Serial CPU vs. GPU

Speedup	**LHS**	**RHS**	**Solve**	**Update**	**Total**
ex05	39.1x	89.9x	17.5x	5.8x	74.1x
ex06	227.0x	378.2x	95.6x	62.1x	509.3x
ex10	267.4x	770.3x	137.8x	53.6x	599.8x
ex20	292.2x	818.9x	165.7x	105.0x	640.7x

V. Conclusion

This work of porting a two-dimensional, real gas hypersonic flow solver was a pathfinding exercise for which we've developed a systematic approach for accelerating legacy Fortran-based CFD solvers. Using the OCCA abstraction library, this process begins with translating each Fortran compute kernel into the OCCA Kernel Language, maintaining the file structure of the legacy codebase. In the Fortran-to-OKL conversion procedure, we systematically convert all Fortran `do-loops` to C-style `for-loops`, replace function input parameter declarations containing multi-index arrays into equivalent array-based pointers using the `@dim` attribute, and replace the Fortran 1-based indexing with C-language 0-based indexing. After verifying the correctness of serial-threaded execution, we add threaded parallelism using @outer and @inner attributes, corresponding to thread-block and thread parallelism, respectively. Lastly, we explore shared and thread-exclusive memory for achieving high performance on specific GPU architectures. During kernel development and optimization, we leverage profiling tools (e.g., NVIDIA Nsight) to determine if the kernel is compute-bound or memory-bandwidth bound. For the latter, our workflow uses the guiding principle of getting 80% or better of the memory bandwidth of the computing device. As a result of this workflow, we were able to port the adjoint capability (in part) – 42 functions with approximately 4,500 lines across 11 files – in three working days. Future work will focus on developing a three-dimensional flow solver capability with increasing numbers of chemical species.

Appendix A

```fortran
SUBROUTINE calctransport(...,Prm,Prmb,Kface,Dface)! input parameters
  USE meshvar
  USE colorvar
  USE state
  IMPLICIT NONE

  ! ... input variables ... !
  REAL(8), INTENT(IN) :: Prm(nvars,nelem)
  REAL(8), INTENT(IN) :: Prmb(nvars,nbndface)
  REAL(8), INTENT(OUT):: Kface(NENERGY+1,nface)
  REAL(8), INTENT(OUT):: Dface(NSPECIES+1,nface)

  ! local variables
  INTEGER :: g,f2,f,e1,e2,e
  REAL(8) :: Prmf(nvarS)
  REAL(8) :: mu,Ks(nenergy+1),Ds(nspecies+1),Cp
  REAL(8) :: mufactor,Kfactor,Dfactor

  mufactor = 1.D0/Re
  Kfactor = rhoinf*Cpinf*Tinf/(Pinf*Re*Prandtl)
  Dfactor = 1.D0/(Re*Schmidt)

  DO g = 1,grp
     !$omp parallel do private(f,e1,e2,Prmf,mu,Ks,Ds)
     DO f2 = grppoint(g),grppoint(g+1)-1
        f = fpoint(f2)
        e1 = fc(1,f)
        e2 = fc(2,f)
        Prmf(:)=0.5D0*(Prm(:,e1)+Prm(:,e2))

        ! calculate collision transport
        call collisiontransport(Prmf,...,Ks,Ds)

        Kface(:,f) = Ks(:)*Kfactor
        Dface(:,f) = Ds(:)*Dfactor
     END DO
     !$omp end parallel do
  END DO

  ! ... Boundary Transport Calculation ... !
END SUBROUTINE calctransport
```

Listing 8 Fortran 90 pseudo-code implementation of the transport calculation with OpenMP pragmas.

```
@kernel
void calctransport(// ... input parameters ... //,
                  @restrict const double *prm,
                  @restrict const double *prmb,
                  @restrict       double *kface,
                  @restrict       double *dface) {
    /* create Fortran-like arrays with multi-indexing */
    const double @dim(NVAR,nelem) *Prm = prm;
    const double @dim(NVAR,nbndface) *Prmb = prmb;
    double @dim(NENERGY+1,nface) *Kface = kface;
    double @dim(NSPECIES+1,nface) *Dface = dface;
    // ...

    /* thread-block parallelism over face groups */
    for (int f2 = startgroup; f2 < endgroup; f2 += p_Nblock; @outer) {

        // thread-local memory
        @exclusive double Ks[NENERGY + 1];
        @exclusive double Ds[NSPECIES + 1];
        @exclusive double Prmf[NVAR];

        // shared memory
        @shared double s_Dinf[NSPECIES + 1];
        @shared double s_ismasses[NSPECIES + 1];
        @shared double s_free_stream_vals[FS_SIZE];
        // ...

        /* thread-level parallelism over faces in face-block */
        for (int t = 0; t < p_Nblock; ++t; @inner) {
            const int f2ind = f2 + t; // face index

            if (f2ind < endgroup) {
                const int f = fpoint[f2ind];
                const int e1 = fc(0,f);       // shift for C-based Index
                const int e2 = fc(1,f);       // shift for C-based Index
                for(int i = 0; i < NVAR; ++i) Prmf[i] = 0.5*(Prm(i,e1)+Prm(i,e2));

                // ... load constants into shared memory ... //
                /* calculate collision transport */
                const double mu = collisiontransport(Prmf,...,Ks,Ds);

                for(int i = 0; i < NENERGY+1; ++i) Kface(i,f) = Ks[i]*Kfactor;
                for(int i = 0; i < NSPECIES+1; ++i) Dface(i,f) = Ds[i]*Dfactor;
            }
        }
    }

    /* ... Boundary Transport Calculation ... */
}
```

Listing 9 OCCA Kernel Language pseudo-code implementation of the transport calculation.

Acknowledgments

This work was supported by NASA STTR grant 80NSSC21C0209 and in part by the National Science Foundation, Office of Advanced Cyberinfrastructure (OAC), Computer and Information Science and Engineering Research Initiation Initiative (CRII), under Award Number 2348394. Computing time was provided by the NCAR-Wyoming Supercomputing Center (NWSC) and the University of Wyoming Advanced Research Computing Center (ARCC). The authors thank Asitav Mishra, Enrico Fabiano, Zhi Yang, Mohammad Zubair, and Boris Diskin for their guidance, feedback, and contributions. The authors would also like to thank Kris Rowe of Argonne National Laboratory for his mentorship as part of the 2021 Argonne GPU Hackathon.

References

[1] Wienke, S., Springer, P., Terboven, C., and an Mey, D., "OpenACC—first experiences with real-world applications," *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings 18*, Springer, 2012, pp. 859–870.

[2] Chandrasekaran, S., and Juckeland, G., *OpenACC for programmers: concepts and strategies*, Addison-Wesley Professional, 2017.

[3] Dagum, L., and Menon, R., "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, Vol. 5, No. 1, 1998, pp. 46–55.

[4] Beyer, J. C., Stotzer, E. J., Hart, A., and de Supinski, B. R., "OpenMP for accelerators," *OpenMP in the Petascale Era: 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, June 13-15, 2011. Proceedings 7*, Springer, 2011, pp. 108–121.

[5] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, Vol. 74, No. 12, 2014, pp. 3202–3216.

[6] Trott, C. R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D. S., Ibanez, D., et al., "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 33, No. 4, 2021, pp. 805–817.

[7] Hornung, R. D., and Keasler, J. A., "The RAJA portability layer: overview and status," 2014.

[8] Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., Pearce, O., Robinson, P., Ryujin, B. S., and Scogland, T. R., "RAJA: Portable performance for large-scale scientific applications," *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*, IEEE, 2019, pp. 71–81.

[9] Medina, D. S., St-Cyr, A., and Warburton, T., "OCCA: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[10] Medina, D., "OKL: A unified language for parallel architectures," Ph.D. thesis, Rice University, 2015.

[11] Chalmers, N., Karakus, A., Austin, A. P., Swirydowicz, K., and Warburton, T., "libParanumal: a performance portable high-order finite element library," , 2022. doi:10.5281/zenodo.4004744, URL `https://github.com/paranumal/libparanumal`, release 0.5.0.

[12] Abdi, D. S., Wilcox, L. C., Warburton, T. C., and Giraldo, F. X., "A GPU-accelerated continuous and discontinuous Galerkin non-hydrostatic atmospheric model," *The International Journal of High Performance Computing Applications*, Vol. 33, No. 1, 2019, pp. 81–109.

[13] Abdi, D. S., Giraldo, F. X., Constantinescu, E. M., Carr, L. E., Wilcox, L. C., and Warburton, T. C., "Acceleration of the IMplicit–EXplicit nonhydrostatic unified model of the atmosphere on manycore processors," *The International Journal of High Performance Computing Applications*, Vol. 33, No. 2, 2019, pp. 242–267.

[14] Kirby, A. C., and Mavriplis, D. J., "Gpu-accelerated discontinuous galerkin methods: 30x speedup on 345 billion unknowns," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2020, pp. 1–7.

[15] Fischer, P., Kruse, J., Mullen, J., Tufo, H., Lottes, J., and Kerkemeier, S., "Nek5000: Open source spectral element CFD solver," *Argonne National Laboratory, Mathematics and Computer Science Division, Argonne, IL, see https://nek5000. mcs. anl. gov/index. php/MainPage*, Vol. 2, 2008.

[16] Fischer, P., Kerkemeier, S., Min, M., Lan, Y.-H., Phillips, M., Rathnayake, T., Merzari, E., Tomboulides, A., Karakus, A., Chalmers, N., and Warburton, T., “NekRS, a GPU-accelerated spectral element Navier–Stokes solver,” *Parallel Computing*, Vol. 114, 2022, p. 102982. doi:https://doi.org/10.1016/j.parco.2022.102982, URL `https://www.sciencedirect.com/science/article/pii/S0167819122000710`.

[17] Brown, J., Abdelfattah, A., Barra, V., Beams, N., Camier, J.-S., Dobrev, V., Dudouit, Y., Ghaffari, L., Kolev, T., Medina, D., et al., “libCEED: Fast algebra for high-order element-based discretizations,” *Journal of Open Source Software*, Vol. 6, No. 63, 2021, p. 2945.

[18] Anderson, J. D., *Hypersonic and high temperature gas dynamics*, American Institute of Aeronautics and Astronautics, Reston, VA, 2019.

[19] Lockwood, B. A., “Gradient-Based Approaches for Sensitivity Analysis and Uncertainty Quantification within Hypersonic Flows,” , 2012. PhD Thesis, University of Wyoming, 2012.

[20] Lockwood, B., and Mavriplis, D., “Gradient-based methods for uncertainty quantification in hypersonic flows,” *Computers & Fluids*, Vol. 85, 2013, pp. 27–38.

[21] Gnoffo, P. A., and Cheatwood, F. M., “User’s Manual for the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA),” 1996.

[22] Wright, M. J., White, T., and Mangini, N., “Data Parallel Line Relaxation (DPLR) Code User Manual: Acadia-Version 4.01. 1,” 2009.

[23] 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., Rumsey, C. L., Thomas, J. L., Thompson, K. B., and Wood, W. A., “FUN3D Manual: 13.6,” , October 2019. NASA/TM-2019-220416.

[24] Candler, G. V., Johnson, H. B., Nompelis, I., Subbareddy, P. K., Drayna, T. W., Gidzak, V., and Barnhardt, M. C., “Development of the US3D Code for Advanced Comppressible and Reacting Flow Simulations,” *53rd AIAA Aerospace Sciences Meeting*, 2015. AIAA Paper 2015–1893.

[25] 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.” 2018. Tenth International Conference on Computation Fluid Dynamics ICCFD10.

[26] MacLean, M., and White, T., “Implementation of Generalized Minimum Residual Krylov Subspace Method for Chemically Reacting Flows,” *50th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, 2012. AIAA Paper 2015–1893.

[27] Kirk, B. S., Stogner, R. H., Bauman, P. T., and Oliver, T. A., “Modeling hypersonic entry with the fully-implicit Navier Stokes (FIN-S) stabilized finite element flow solver,” *Computers & Fluids*, Vol. 92, 2014, pp. 281 – 292. doi:https://doi.org/10.1016/j.compfluid.2013.10.003, URL `http://www.sciencedirect.com/science/article/pii/S0045793013003861`.

[28] Dunn, M. G., and Kang, S., “Theoretical and experimental studies of reentry plasmas,” Tech. rep., NASA, 1973.

[29] Park, C., “Nonequilibrium hypersonic aerothermodynamics,” 1989.

[30] Liou, M.-S., “A sequel to AUSM, Part II: AUSM+-up for all speeds,” *Journal of computational physics*, Vol. 214, No. 1, 2006, pp. 137–170.

[31] Edwards, J. R., “A low-diffusion flux-splitting scheme for Navier-Stokes calculations,” *Computers & Fluids*, Vol. 26, No. 6, 1997, pp. 635–659.

[32] Mavriplis, D., “Revisiting the least-squares procedure for gradient reconstruction on unstructured meshes,” *16th AIAA computational fluid dynamics conference*, 2003, p. 3986.

[33] MAVRIPLIS, D. J., *SOLUTION OF THE TWO-DIMENSIONAL EULER EQUATIONS ON UNSTRUCTURED TRIANGULAR MESHES (AIRFOILS, TRANSONIC)*, Princeton University, 1987.

[34] Edwards, J., “A low-diffusion flux-splitting scheme for Navier-Stokes calculations,” *Oceanographic Literature Review*, Vol. 1, No. 45, 1998, p. 187.

[35] Olynick, D. R., “A new LU-SGS flow solver for calculating reentry flows,” Ph.D. thesis, 1992. URL https://uwyo.idm.oclc.org/login?url=https://www.proquest.com/dissertations-theses/new-lu-sgs-flow-solver-calculating-reentry-flows/docview/304020838/se-2, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-07-18.

[36] Hassan, B., “Thermo-chemical nonequilibrium effects on the aerothermodynamics of hypersonic vehicles,” Ph.D. thesis, 1993. URL https://uwyo.idm.oclc.org/login?url=https://www.proquest.com/dissertations-theses/thermo-chemical-nonequilibrium-effects-on/docview/304071040/se-2, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2024-07-18.

[37] Hänel, D., Schwane, R., and Seider, G., “On the accuracy of upwind schemes for the solution of the Navier-Stokes equations,” *8th Computational Fluid Dynamics Conference*, 1987, p. 1105.

[38] v3.0.0, E., 2023. doi:10.11578/E3SM/dc.20240930.1, URL https://e3sm.org, gitHub Repository: https://github.com/E3SM-Project/E3SM, BSD-3-Clause.

[39] Medina, D. S., St-Cyr, A., and Warburton, T., “OCCA: A unified approach to multi-threading languages,” *arXiv preprint arXiv:1403.0968*, 2014.

[40] Medina, D., “OKL: A unified language for parallel architectures,” , 2015. PhD Thesis, Rice University, 2015.

[41] Luo, W., Fan, R., Li, Z., Du, D., Wang, Q., and Chu, X., “Benchmarking and Dissecting the Nvidia Hopper GPU Architecture,” , 2024. URL https://arxiv.org/abs/2402.13499.

[42] Cuthill, E., and McKee, J., “Reducing the Bandwidth of Sparse Symmetric Matrices,” *Proceedings of the 1969 24th National Conference*, Association for Computing Machinery, New York, NY, USA, 1969, p. 157–172. doi:10.1145/800195.805928, URL https://doi.org/10.1145/800195.805928.

[43] 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.

[44] Carson, E., and Higham, N. J., “Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions,” *SIAM Journal on Scientific Computing*, Vol. 40, No. 2, 2018, pp. A817–A847. doi:10.1137/17M1140819.

[45] “CUDA Pro Tip: Minimize the Tail Effect,” https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/, 2014. Accessed: 2021-11-12.