

GPU Acceleration of VULCAN-CFD

Matthew O'Connell, Kyle Thompson, Jeffery White, Robert Baurle
Clark Pederson, and Andrew Hinkle

NASA Langley Research Center, Hampton, VA 23682, USA

ABSTRACT

This work presents a comprehensive overview of recent advancements in the application of Graphics Processing Units (GPUs) to accelerate the NASA-developed VULCAN-CFD code for hypersonic flow simulations. The unstructured solver in VULCAN-CFD is undergoing a significant rewrite from modern Fortran to C++, enabling its execution on both GPUs and CPUs through the utilization of Kokkos, a programming model for performance portability developed by Sandia National Labs. The paper outlines some modifications that were made in the original implementation of VULCAN-CFD in order to harness the computational power of GPUs. Finally, we demonstrate performance improvements achieved through GPU acceleration. The accelerated code throughput on one GPU is shown to match approximately 200 CPU cores for both single species and multispecies reacting cases.

INTRODUCTION

The landscape of Computational Fluid Dynamics (CFD) is undergoing a transformative revolution, marked by a significant shift towards harnessing the power of Graphics Processing Units (GPUs) for high-performance computing. This paradigm shift reflects not only the increasing demand for faster and more efficient simulations but also the recognition of GPUs as a pivotal tool to unlock unprecedented computational capabilities. The paradigm shift resonates with the broader CFD community, underlining the industry's collective recognition of GPUs as a game-changing force. Across the CFD community, established government codes such as FUN3D [1] and OVERFLOW [2], open-source pioneers like Palabos [3] and OpenFOAM [4], and industry giants, such as Ansys [5], are all converging on this trajectory of modernization.

In this paper, we present the latest developments in VULCAN-CFD, showcasing how the solver has embraced GPU acceleration and modern software architecture while remaining faithful to its core principles of accuracy and reliability. In this work, we describe an ongoing effort in porting the unstructured solver capabilities in VULCAN-CFD from Fortran to C++ and adopting Kokkos [6] as a performance portability model so that we can leverage GPU hardware. These efforts represent the first year of a two year plan to bring GPU simulation capabilities to production simulations using VULCAN-CFD. The primary focus of this effort has been to perform a risk reduction study to assess the viability of our modernization approach before migrating the breadth of the VULCAN-CFD capabilities. We have implemented a prototype solver containing a subset, or vertical slice, of the VULCAN-CFD unstructured simulation capabilities in C++ with the plan that if major hurdles were encountered in porting to GPUs we would have uncovered those challenges early and with a small and nimble code base that would facilitate a pivot in our strategies quickly.

The forthcoming sections of this paper will provide a brief overview of our journey, including an exploration of the methodologies used in our GPU-accelerated VULCAN-CFD, and any significant departures from the traditional Fortran codebase. Finally, we will delve into performance results obtained during the testing and validation phases, shedding light on the impressive throughput and efficiency exhibited by the GPU-accelerated solver for both single- and multispecies reacting problems.

METHODS

In this section, we delve into the innovative design choices and transformations that distinguish the GPU-accelerated VULCAN-CFD solver from its Fortran predecessor. These strategic departures reflect the solver's evolution towards improved performance, modularity, and adaptability.

Performance Portable Programming Model

The overwhelming majority of the VULCAN-CFD 600 thousand lines of code is written in Fortran 90. The first hurdle we encountered was the inability to directly program in Fortran for GPUs in a way that is supported by all GPU hardware. Figure 1 shows a table of different GPU programming paradigms and the level of support each paradigm has with the different GPU vendors (adapted from reference [7]). Each vendor has their own preferred programming paradigm, CUDA from NVIDIA, HIP from AMD, and SYCL from Intel. NVIDIA's CUDA is currently the only vendor paradigm with Fortran support; however, the three major GPU vendors all support a C++ programming model.

Since VULCAN-CFD is such a large code base, we cannot plan on multiple rewrites to support different hardware vendors. This ruled out using any of the vendor-specific programming paradigms. A vendor agnostic programming paradigm was therefore desirable. The vendor agnostic programming models of OpenACC and OpenMP were initially considered as they had Fortran language support, and success stories could be found in the literature for OpenACC using structured grids, including many academic efforts [8] and, more recently, production CFD solvers from OVERFLOW [2]. However, for CFD solvers using unstructured grids, although a few academic examples exist [9], no CFD GPU-enabled capabilities could be identified that focus on production engineering work.

When we started the effort, major government CFD codes LAURA, SPARC [10], and FUN3D [1] had all chosen C++ as the underlying language for GPU programming with LAURA and SPARC both selecting Kokkos [6] as a vendor agnostic programming paradigm. Motivated by the long collaboration between LAURA and VULCAN-CFD, we also selected C++ and Kokkos for the GPU prototype solver.

	CUDA		HIP		SYCL		Standard		OpenMP		OpenACC		Kokkos	
	C	F	C	F	C	F	C	F	C	F	C	F	C	F
NVIDIA	●	●	■	✗	▲	✗	●	●	●	●	●	●	▲	✗
AMD	■	✗	●	✗	■	✗	▲	✗	●	●	▲	▲	▲	✗
INTEL	■	✗	✗	✗	●	✗	●	●	●	●	✗	✗	▲	✗

●	Full vendor support	▲	Comprehensive support, not from vendor
●	Partial vendor support	■	Indirect vendor support
✗	Very limited or no support		

Figure 1: The current state of programming models, compatibility by language (C/C++ or Fortran) and different GPU vendors.

Overview of VULCAN-CFD Prototype Solver

VULCAN-CFD currently is approximately 600 thousand lines of Fortran. Much of the VULCAN-CFD code base is not performance-critical and is instead used for “pre” and “post” processing. During those steps user input is parsed to prepare the solver for a simulation, and the simulation results are processed to extract engineering output, respectively. The major performance critical part of VULCAN-CFD occurs in the nonlinear solver iteration. For each nonlinear iteration, VULCAN-CFD forms and solves a linear problem:

$$\left(V \frac{dQ}{dt} I + \frac{dR(Q)}{dQ} \right) \Delta Q = R(Q), \quad (1)$$

where V is the volume of a given cell, Q are the conserved variables of density, momentum, and energy, and R is the discrete evaluation of the Reynolds-averaged Navier-Stokes equations.

The nonlinear solver has 3 major phases: calculating $R(Q)$, calculating $\left(V \frac{dQ}{dt} I + \frac{dR(Q)}{dQ} \right)$ and then solving the linear system for ΔQ . We will refer to these three phases as RHS (right hand side), LHS (left hand side), and linear solve (linear system). Equation 1 is solved for every compute cell in a simulation mesh. Large simulations can take advantage of multiple CPU cores (or multiple GPUs) by partitioning the mesh and mapping each partition to compute hardware (either a single CPU core, or a single GPU). Both the RHS and the linear phases require communication between partitions. The LHS only uses a first-order approximation of the Jacobian allowing the formation of an efficient sparse linear system without requiring any additional cross partition communication.

A Prototype Implementation

VULCAN-CFD has many different modeling options that change the implementation of each of the three phases. For example, VULCAN-CFD currently supports 3 inviscid flux schemes, 5 variable reconstruction schemes, 6 limiters, 10 gradient advection options, and multiple turbulence models. The differences in these options range from nuanced to critical depending on the engineering problem being solved. However, these decisions often have only minor influence on the performance of the solver. Since we wish to quickly assess whether there were any major hurdles in porting VULCAN-CFD to GPUs, we opted to create a prototype implementation that would be a vertical slice through the major performance-critical components of the solver.

A meaningful selection of the total available options in VULCAN-CFD is implemented in the prototype solver to help guide the new software architecture and test performance. Once a complete vertical slice of the solver was implemented, the remaining features can be implemented by copying the patterns used in the prototype. Appendix A contains a list of options in VULCAN-CFD's unstructured solver and status of each option in regards to the prototype implementation as of this writing in mid October 2023. This paper will present performance results of this vertical slice and the conclusions of this risk reduction study.

A (mostly) faithful recreation

In our endeavor to transition VULCAN-CFD to a GPU-accelerated architecture, we made a conscious decision to preserve the core discretization choices that have defined the original Fortran version. This continuity ensures that the solver retains its identity and continues to provide the reliability and accuracy that users have come to expect. However, developing the prototype created an opportunity to reevaluate some implementation choices from the existing VULCAN-CFD. In this section, deviations from the original implementation are highlighted.

Unifying parallelism: the global ghost model The original unstructured solver implementation grew as a modification of the existing structured solver. Where this is most evident is in the partitioning model for distributed memory parallelism.

The structured solver primarily followed a face-based communication pattern where cells in a given i, j , or k direction needed face neighbors along those directions. For distributed memory parallelism this meant communicating only face neighbors. The original implementation of the unstructured solver relied on Green-Gauss gradients which also only required face-neighbor cells [11]. Therefore, much of the MPI communication used for the structured solver could be easily adapted for the unstructured solver.

However, the unstructured discretization has evolved to rely more on Least Squares (LSQ) approaches for gradient reconstruction [12]. For LSQ-based gradients, the LSQ stencils require both face and node neighbors of varying degrees. A separate communication pattern was implemented to handle this different kind of data dependence since it did not fit the pattern of communicating face-neighbor data. This additional communication pattern, referred to as U2U-nodes (unstructured-to-unstructured node stencils) relied on distinct MPI buffers and derived types to store a local copy of nonlocal cell data. That meant that during the nonlinear solve step **both** face-based communication and u2u-node communication had to be performed. Much of that communication is redundant since cells near partition boundaries will be both face neighbors to some cells, and node neighbors to others.

The prototype solver, developed for GPUs, instead unifies all distributed memory communication using a global ghost model. In this model, each cell is uniquely “owned” by a specific partition just as in the previous model. However, instead of only storing the owned cells on a partition, the mesh is extended to encompass all cells needed by the least squares stencils. Each partition maintains complete ghost copies of these cells and only differentiates ghost cells from owned cells using an array that stores the owning partition number for each cell. All cells that are stored on a partition are considered “resident” to the partition, while only some of the resident cells will be owned by that partition. This shift simplifies MPI communication, consolidating various communication exchanges (such as those for gradients, solution reconstruction, and face data) into a single synchronization type. Furthermore, once a field is synced it can be used for flux evaluation, gradient reconstruction, and limiter calculation without any additional communication.

The global ghost cell model optimizes parallelism and communication efficiency, particularly in multi-GPU configurations. It streamlines data exchange between partitions, reducing the overhead associated with multiple communication exchanges. This model enhances scalability but otherwise has no impact on the solution. In the results section we will see that this unifying communication pattern enables increased scalability due to the reduction in communication cost.

Linear Solver Solving the linear problem from equation 1 requires solving a sparse block linear system. VULCAN-CFD has historically used variations of Symmetric Gauss-Seidel (SGS) for solving this sparse block matrix on each unstructured partition and lagged update across partitions. Unfortunately, neither SGS, nor the more traditional Gauss-Seidel (GS) can be applied directly to GPUs. This is because both GS and SGS have strong data dependence across rows of the linear system. The GS update from one row is dependent on the result of the previous row. In essence, GS is a serial algorithm. Therefore, we have adopted using a MultiColored Gauss-Seidel (MCGS) following the work on Zubair et al. [13]. The MCGS algorithm was implemented in Enigma [14], a linear and nonlinear solver framework being used across VULCAN-CFD [15], LAURA [14], and HyperSolve [16].

With MCGS, first a coloring algorithm is applied to each block row of the matrix so that every row is not assigned the same color as any column that appears in the row sparsity. In VULCAN-CFD, the sparsity of the Jacobian matrix is based on the first order advection stencil, i.e., each cell face neighbor. This means that the coloring algorithm assigns a color to each cell, such that no two adjacent cells have the same color. Figure 2 shows an example of coloring applied to cells of a test compression corner grid. The coloring algorithm is applied during startup and is run across MPI partitions to ensure a globally consistent coloring scheme. During the linear solve, every cell assigned to a given color will not directly influence any other cell of the same color. Therefore, every cell of a given color can be updated simultaneously. The MCGS algorithm is summarized in Algorithm 1.

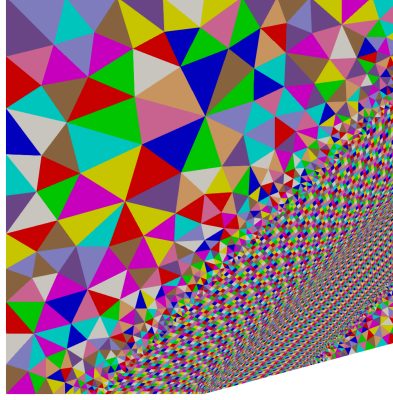


Figure 2: Result of coloring algorithm applied to a test compression corner grid.

Algorithm 1: MultiColored Gauss-Seidel

Input : Global Sparse Block Matrix A , Global Block Vector b

Output: Global Block Vector x

```

let  $x = \text{ones}$ 
let  $D = \text{diag}(A)$ 
let  $O = \text{offdiag}(A)$ 
let  $L, U = \text{factor}(D)$ 

for each Gauss-Seidel iteration sequentially do
  for  $c$  in colors sequentially do
    forall rows with color  $c$  parallel do
       $b = b - O x$ 
       $x = \text{solve}(L, U, b)$ 
    end
    if using multiple MPI partitions then
       $\text{sync}(x)$ 
    end
  end
end

```

Verification and Validation of the Prototype GPU Solver To maintain the fidelity of the new version of VULCAN-CFD, we have implemented a rigorous regime of continuous integration (CI) testing. This ongoing testing process ensures that the new version consistently produces results that closely align with the original Fortran code across a spectrum of canonical problems. This battery of tests encompasses unit-tests, Method of Manufactured Solutions (MMS) tests to ensure design order of accuracy, and verification tests such as those drawn from the NASA Turbulence Modeling Resource [17].

Figure 3 shows one example of a test that is performed on VULCAN-CFD. This example shows a converged flat plate verification case for the SA model. Previously validated results are compared every night against those obtained from the most up-to-date development version of VULCAN-CFD. These results are also published to a developer facing website. The output artifacts (console output, post-processing files, and images) are saved with an accompanying time stamp in case any failures are detected. These Functional Equivalency Tests (FETs) are not the same as typical regression testing where perhaps only a few iterations of a solver are executed. The FETs are end-to-end tests that are converged to machine zero every time they are run. The commitment to CI guarantees that the new version of VULCAN-CFD maintains its legacy of accuracy and reliability while embracing the benefits of GPU acceleration and modern architecture.

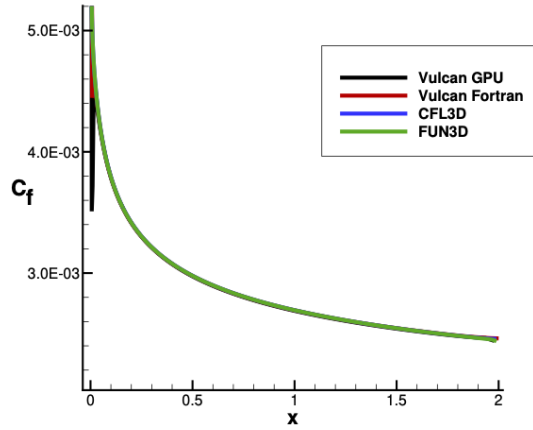


Figure 3: An example turbulence model verification test run nightly to ensure accuracy of the GPU implementation. The graph shows skin friction vs downstream distance on a flat plate.

RESULTS

In our pursuit of optimizing VULCAN-CFD for GPU acceleration, we conducted a series of performance tests representative of steady-state RANS simulations. All the performance tests are based on timing measurements of the different phases of a nonlinear iteration (RHS, LHS, and linear solve). Grids of varying sizes were generated using *refine* [18] to scale up or down grids that were adapted using the Sketch-2-Solution workflow [15],[19]. All the grids are tetrahedral 3D grids.

Two different simulation cases are presented: a single species, calorically perfect gas, and a 7 species thermally perfect gas mixture utilizing a 7-reaction finite rate chemical kinetics model based on hydrogen reaction chemistry. For both cases, GPU and CPU throughput is measured, and conversion rates computed to determine the number of CPU cores it takes to match the throughput of a single GPU. The GPU compute node used for testing consisted of 8x NVIDIA A100 80GB [20] GPUs. CPU performance was tested using up to 16 compute nodes, each equipped with 2x 20 core Intel(R) Skylake Xeon(R) Gold 6148 CPUs [21]. Additional performance tests include the strong and weak performance scalability. MPI is used for all cross process communication. Each CPU core is assigned an MPI rank when running in CPU mode, and each A100 GPU is assigned to a unique MPI rank in GPU mode.

We conducted performance tests using two versions of the code: the original Fortran version on CPU and the new C++ version on both CPU and GPU. Initially, we ran a few tests on the Fortran version to confirm its performance similarity to the C++ version. Having established this equivalence, we proceeded to conduct the bulk of our performance assessments on the C++ version, taking full advantage of its GPU acceleration capabilities.

Perfect gas test case

To evaluate VULCAN-CFD for GPU acceleration, we conducted a series of tests for simulations of perfect gas, steady-state RANS using the SA turbulence model [22]. Single species RANS is a common use case for VULCAN-CFD and represents the most mature physical modeling type currently implemented in the prototype solver. The geometry for this test case is inspired by NASA's X-43A hypersonic vehicle. The actual vehicle geometry is taken from an Engineering Sketch Pad (ESP) tutorial case [23] and, as such, is not an accurate representation of the test vehicle.

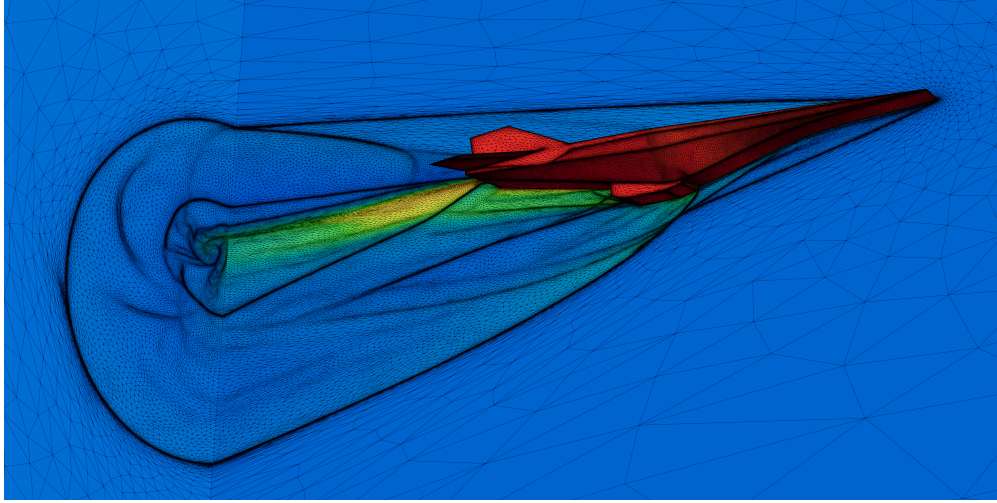


Figure 4: Sketch-2-Solution adapted grid X-43A perfect gas simulation.

As part of a previous effort, the typical Sketch-2-Solution workflow [15] was used with VULCAN-CFD to generate 17 grids, the final grid reaching a size of just under 18 million cells. The contour plot of the static temperature obtained on the 18 million cell grid is shown in Figure 4. For the following test cases, grids smaller than 18 million cells were taken from the original Sketch-2-Solution adaptation trajectory. Since only grid scaling performance metrics were desired, grids larger than 18 million cells were created by using *refine* [18] to scale up the grid size while maintaining the relative spacing from the 18 million cell grid.

Performance Throughput The first test case is a throughput test. This test runs the solver on grids of varying sizes while measuring the execution time for the LHS, RHS, and linear solve phases of the nonlinear iteration. The throughput is measured as the time it takes to complete each phase of the iteration, divided by the number of cells in the grid:

$$\text{GPU Throughput} = \frac{\text{Cell Count}}{\text{Execution Time}}$$

Compute hardware is made of lots of individual components that can potentially bottleneck with different sized grids. For example, a single GPU is comprised of a very large number of low performance compute cores* and, if the grid is too small, only a portion of these compute cores may be utilized. By varying the grid size, it can be determined how big a grid should be to efficiently leverage either a single GPU or a full CPU compute node. Figure 5 shows throughput for grids of varying sizes using a single GPU. The GPU is capable of computing just over 3.2 million cells per second once the grid is suitably large. The throughput metric stays relatively constant for grid sizes above approximately 4 million cells, though throughput remains high down to approximately 1 million cells. Below 1 million cells, there is not enough work to efficiently load up the GPU.

Figure 6 shows the relative throughput for each phase of the nonlinear solve. The relative throughput is obtained for each grid by normalizing the throughput of each phase by the throughput that phase achieved on the largest grid. We see that the LHS, and linear solve phases reach near peak throughput at around 1 million cells, while it takes closer to 4 to 5 million cells before the RHS evaluation reaches its peak throughput. We can also see that the linear solve phase shows stable throughput above 5 million cells but achieves less than 75% of this performance on the smallest grid (consisting of approximately 113 thousand cells). We see that the total throughput on the smallest grid in the series is achieving approximately 85% of the performance observed on the largest grid.

*An A100 has 6912 CUDA cores.

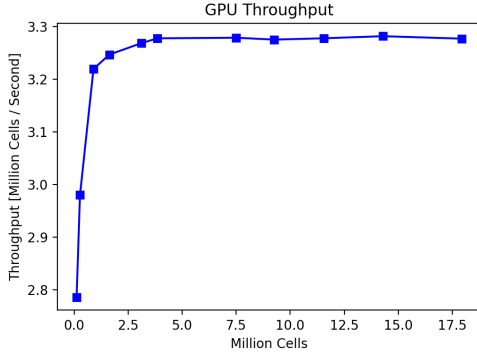


Figure 5: Total throughput for a single GPU with varying grid size.

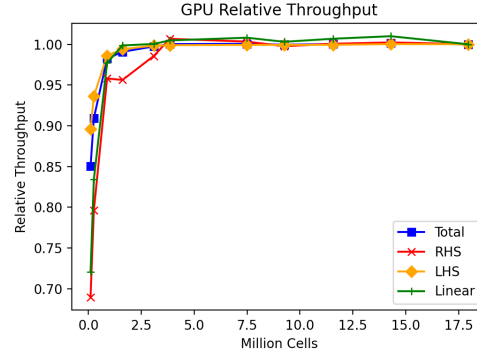


Figure 6: Relative throughput for each phase of the nonlinear with varying grid size.

Figure 7 shows a similar throughput test conducted on a single CPU-based compute node consisting of 40 Intel Skylake cores. There are many other configurations of CPU compute nodes with varying core counts, and modern CPUs can dynamically boost the performance of single core tasks when the thermal load on the CPU is low. Therefore, we opted to use the entire CPU compute node simultaneously and compute throughput normalized per CPU core:

$$\text{CPU Throughput} = \frac{\text{Cell Count}}{\text{Execution Time} \times \text{CPU Core Count}}$$

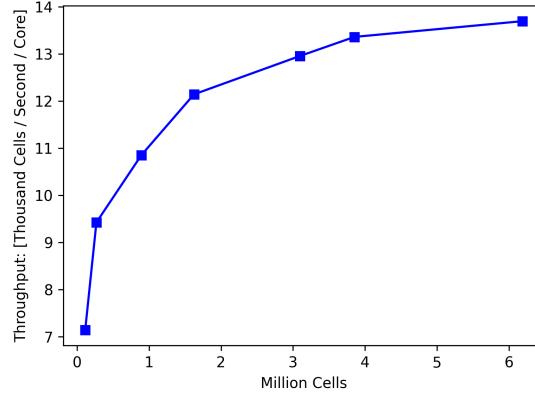


Figure 7: Total throughput per CPU core of the nonlinear iteration.

With the GPU throughput, we observed that the overall throughput reached a peak at around 1 million cells and then the throughput remained nearly constant. However, the CPU throughput does not quickly reach level nor does the throughput remain constant as the GPU did. Instead the throughput continues to increase with grid size up to the 6 million cells. Grid sizes of 7 and 9 million were also run, however the 7 million cell case exhibited a significant drop in throughput and the 9 million cell case ran out of system memory.

The prototype solver running on GPUs can leverage both GPU and system memory simultaneously. The prototype solver allocates the Jacobian matrix and any temporary arrays used by the linear solver only on the GPU device when running in GPU mode. Likewise, many additional arrays used by the prototype solver are never needed on the GPU (for example arrays related to post-processing, and arrays related to debugging and diagnostics). When running on only CPU hardware, all memory allocations must be done in system RAM. This limits the maximum grid size that can fit in memory on a CPU compute node compared

to a GPU. Many of the debugging and diagnostic arrays will be removed from the prototype solver with future development and should help close the gap between the maximum grid that can fit on CPU and GPU compute nodes.

In summary, the GPU was able to show stable performance of approximately 3.275 million cells per second once a large enough grid size was used (approximately 1 million cells or more). A single 80 GB GPU was able to maintain this throughput on grid sizes up to 18 million cells. For the 40 core CPU compute node the throughput was significantly lower reaching a peak of approximately 13.8 thousand cells per core (0.675 million cells per CPU node). The single GPU throughput rate is approximately 220 times the throughput of a single CPU core. The throughput test essentially shows an upper limit of the rate of work each single GPU, or CPU core can perform under idealized settings where the grid size is ideal for the compute hardware and there is no data transfer between the system memory and the GPU or over the network between compute nodes. If the implementation scaled perfectly with additional compute hardware, the throughput number would simply be multiplied by CPU core count (or GPU count) to predict the computational cost.

Strong Scaling Study The next study was conducted to comprehensively assess the strong scaling performance of the VULCAN-CFD prototype solver. Strong scaling studies fix the size of the computational problem and add successively more compute resources to determine how much faster a given problem can be solved. The strong scaling study was conducted using between 1 and 8 GPUs and between 20 and 640 CPU Cores.[†] The grid used for the strong scaling study was the largest used on matching hardware from the throughput test. Therefore, the CPU strong scaling test was run on the 6 million cell grid, and the GPU scaling test was run on the 18 million cell grid.

Figure 8 shows the strong scaling results for an 18 million cell grid on up to 8 A100 GPUs. The Figure shows speedup for each phase of the nonlinear solve, as well as the total speedup measurement. Speedup is computed as

$$\text{Speedup}(N) = \frac{\text{Execution Time on 1 GPU}}{\text{Execution Time on } N \text{ GPUs}}$$

The total nonlinear solver scales well to 2 GPUs but shows little improvement beyond 4 GPUs. The LHS evaluation scales very well all the way to 8 GPUs. Recall that the LHS has no communication cost and, based on the previous throughput results, an 18 million cell grid split into 8 partitions still leaves over 2 million cells on each partition that is well within the range of acceptable LHS throughput. The linear solve phase requires significant cross GPU communication and we see that it scales very poorly even so far as to see a drop in performance from 4 to 8 GPUs. The RHS phase also exhibits poor GPU scaling performance with only very modest throughput gains between 4 and 8 GPUs since the RHS has comparatively low computation and relatively high communication. Furthermore, partitioning the problem for 4 GPUs leaves approximately 4.5 million cells per a single GPU. During the single GPU throughput tests, we observed a drop in RHS performance around 4 million cells. Therefore, we cannot expect to see ideal strong scaling for the RHS on this size grid on 8 GPUs even if the solver had zero communication overhead.

[†]CPU strong scaling tests were run on as little as a single 20 cores of a 40 core CPU node.

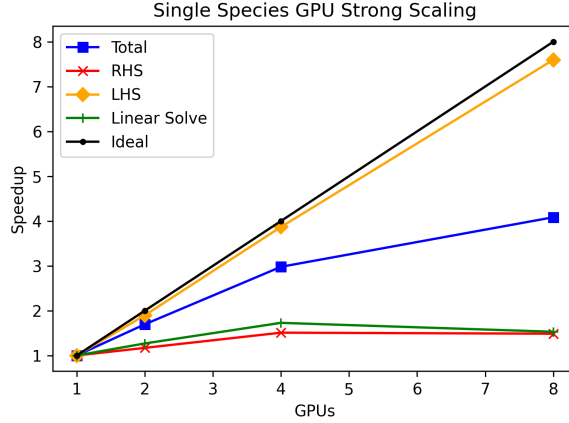


Figure 8: Strong scaling of an 18 million cell grid across varying GPU counts.

Supporting multi-GPUs is a relatively new feature in the prototype solver and further development efforts are expected over the next year. For now, we will move on to comparing a single GPU performance against a multi-CPU strong scaling study to compare the benefit of a single GPU over CPUs for typical RANS cases.

Figure 9 shows the speedup obtained as the result of the strong scaling study across varying CPU core counts. The CPU compute nodes utilized for this study have two 20-core Intel Xeon CPUs for a total of 40 cores per compute node. Each node has a minimum of 96 GB of system memory. The speedup in Figure 9 is computed on a per node basis:

$$\text{Speedup}(N) = \frac{\text{Execution Time on 1 CPU Node}}{\text{Execution Time on N CPU Nodes}}$$

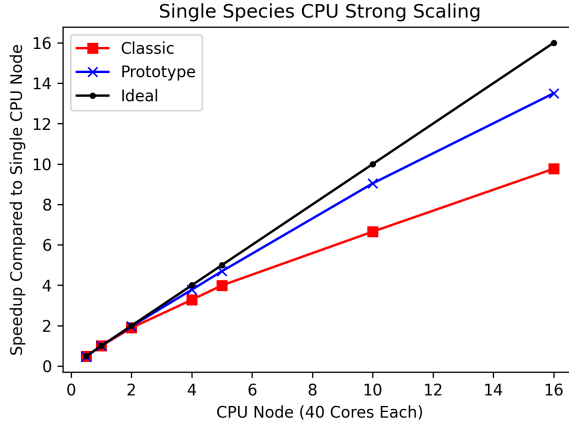


Figure 9: Strong scaling speedup of a 6 million cell grid across varying CPU compute nodes.

Since the existing Fortran implementation of VULCAN-CFD can also run on CPUs, we ran both the “classic” implementation and prototype C++ / Kokkos implementation for this strong scaling study. From Figure 9, we can see that above 2 CPU compute nodes (80 CPU cores) the new prototype implementation exhibits improved strong scaling speedup compared to the “classic” implementation. Improved strong scaling was a goal of the prototype solver and was a driving reason we adopted the unified global ghost cell model.

Figure 10 shows the total execution time of a nonlinear solve step as well as a breakdown of the three phases (RHS, LHS, and linear solve). The new prototype solver is slightly faster across all CPU ranges, ranging from approximately 10% faster on 20 cores to approximately 25% faster at 640 cores. To help understand the performance impact for a single GPU, we also ran the 6 million cell grid using an A100 GPU. The execution time using a single A100 is plotted as a horizontal blue line. For this case, it takes approximately 180 CPU cores for the prototype implementation and 200 CPU cores for the classic implementation to match the performance of the prototype solver running on a single A100 GPU.

Figure 10b shows the execution time of the RHS. Here we see that the prototype solver shows improved scalability at higher core counts compared with the classic solver. The prototype solver on 200 cores approximately matches the execution time of the classic solver at 640 cores. The single GPU is as fast as approximately 500 cores when calculating the RHS using the prototype solver. It is likely that, due to scalability limits for a grid this size, the classic implementation will not scale strongly enough on any number of CPU cores to match the performance of a single A100.

Figure 10c shows the execution time of the LHS. The LHS in the prototype solver is approximately 15% slower than the classic solver on 20 cores and 22% slower at 640 cores. Both implementations used automatic-differentiation to obtain the LHS. It is common for the “classic” solver to compute the Jacobian matrix using analytically derived formulations. However, this option is not yet available in the prototype solver and was not tested.[‡] Again the single GPU execution time is plotted as a horizontal blue line. The relative GPU to CPU performance is the lowest for the LHS phase. The single GPU execution time matches approximately 120 CPU cores for the prototype implementation and 150 cores for the “classic” implementation.

Finally, Figure 10d shows the execution time of the linear solver. Recall that the prototype implementation and the classic implementation use slightly different linear solvers. The classic implementation uses symmetric Gauss-Seidel, while the prototype solver uses a MultiColor Gauss-Seidel because symmetric Gauss-Seidel cannot be implemented using GPUs. The execution times reported include all the communication required for both linear solver implementations, and both linear solvers were configured to run 5 linear solver sweeps. We see that the linear solver is faster in the prototype implementation across all CPU core counts. A single GPU is about as fast as 250 CPU cores for the prototype solver, and approximately 550 cores when comparing to the classic implementation.

Overall, we see that for a RANS of a calorically perfect gas solved with 6 million grid cells the number of CPU cores that a single GPU replaces varies between approximately 120 to 550 cores for each phase of the flow solve process. The overall cost comparison shows consistent results with the throughput test where one GPU is equivalent to approximately 200 CPU cores.

Weak Scaling Study For the final single species RANS performance study we conducted a “weak” scaling study. Weak scaling measures the efficiency when solving larger problems by using more compute resources. In this instance, the 18 million cell grid was scaled up using *refine* to grids of size 33.8 M, 68.3 M, and finally 131.9 M. These grids were run on 2, 4, and 8 GPUs, respectively.[§]

The throughput for each grid was computed and the results are shown in Figure 11 along with the ideal throughput. Ideal throughput on N GPUs is computed as

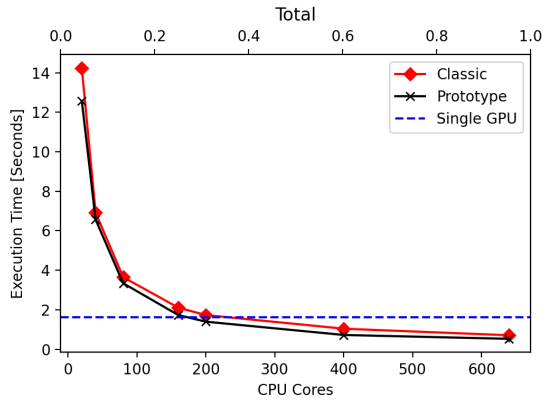
$$\text{Ideal Throughput}(N) = \text{Throughput on 1 GPU} \times N.$$

Efficiency is shown in Figure 12 where efficiency is computed as

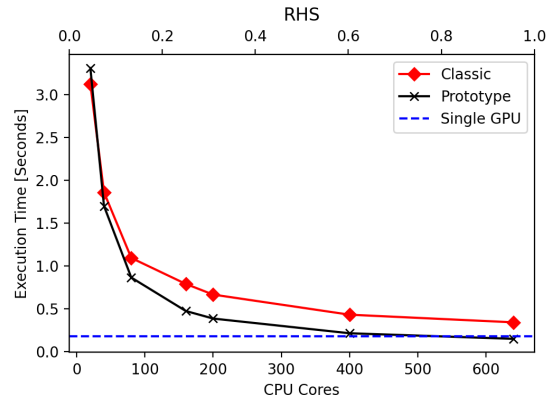
$$\text{Efficiency}(N) = \frac{\text{Throughput}(N)}{\text{Ideal Throughput}(N)}.$$

[‡]The future development plan is outlined in Appendix A.

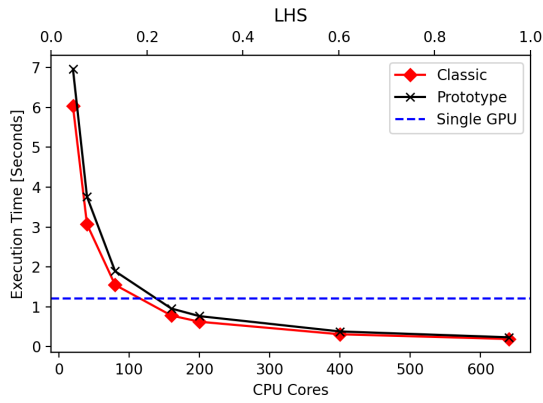
[§]The 18 million cell grid was very near the maximum grid size that could fit on a single GPU. When running a partitioned problem there is some memory overhead to store the ghost cells and communication arrays. Therefore, the grid sizes were increased slightly less than factors of 2, 4, and 8.



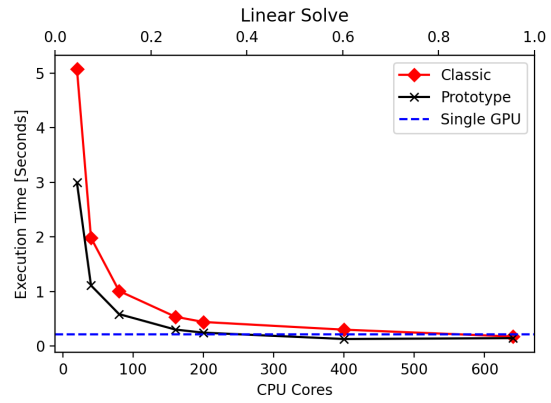
(a) Total nonlinear iteration.



(b) Right hand side formulation phase.



(c) Left Hand Side formulation phase.



(d) Linear solve phase.

Figure 10: Strong scaling study with varying CPU core counts.

The prototype solver does not exhibit ideal weak scaling. The weak scaling efficiency quickly drops to 80% on 2 GPUs and falls to about 70% at 8 GPUs. Despite not maintaining ideal scaling, the solver achieves a throughput of nearly 16 million cells per second on up to 8 GPUs on a grid size of 132 million cells. At this rate, a simulation on a 132 million cell mesh would take approximately 1 day to perform 10 thousand iterations.

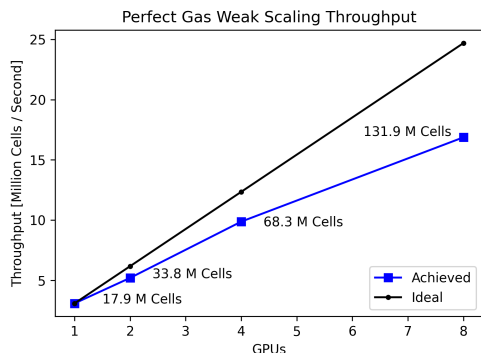


Figure 11: Weak scaling throughput.

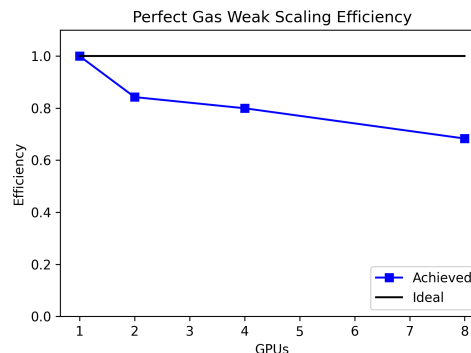


Figure 12: Weak scaling efficiency.

Multispecies Reacting Simulation Performance

The next study is performed to analyze computing performance for chemically reacting cases. An overview of the test problem is shown in Figure 13. The test geometry consists of a long duct with a cavity flameholder and two fuel injectors. The flow into the combustor was at Mach 2.4 with a mix of N₂ and O₂ and the fuel injectors injected hydrogen. A Sketch-2-Solution case was run for a few grid adaptations until the grid reached approximately 5 million cells. Then the final grid was scaled to different mesh sizes and a series of computing performance studies were conducted. For the performance studies, only a few iterations were run in order to obtain performance data.

Throughput Test For the throughput test, a single A100 card was used along with an entire Skylake-based CPU compute node. The largest grid run on the A100 GPU was 5.3 million cells and nearly all of the available 80 GB of GPU memory was utilized running this grid. The largest grid run on the CPU node was 2.7 million cells and nearly consumed all the CPU node's 96 GB of system memory. Figure 14 shows throughput performance on the GPU. At peak throughput (between about 1 and 5 million cells), the prototype solver is capable of completing a full nonlinear iteration at a rate of approximately 750 thousand cells per second. In comparison, Figure 15 shows throughput performance per CPU core. The CPU throughput doesn't appear to have a level throughput curve and instead exhibits improved throughput as the grid size increases all the way through to the largest grid tested. The peak CPU throughput observed is approximately 3.7 thousand cells per second. The A100 GPU shows a throughput that is approximately 200 times that of the peak CPU throughput.

Figure 16 shows each portion of the solver and the total cost normalized by the throughput of the largest grid of 5.3 million cells. This is significantly less than the 18 million cell grid that was able to fit in GPU memory for the single species case. The reduced cell size is primarily due to the Jacobian matrix size being based on 13x13 block matrices vs the 6x6 matrices needed for single species gas with a one equation turbulence model. Note that the large Jacobian matrix size can be mitigated by using a decoupled implicit chemistry methodology [24] for large species cases, or by utilizing mixed precision [25].

We see that the linear solver reaches above 95% throughput at around 1/2 million cells, while the LHS and RHS phases reach 95% throughput at around 1 million cells. These throughput results are consistent with what was observed for the single species, calorically perfect gas case. The parallelism exposed by

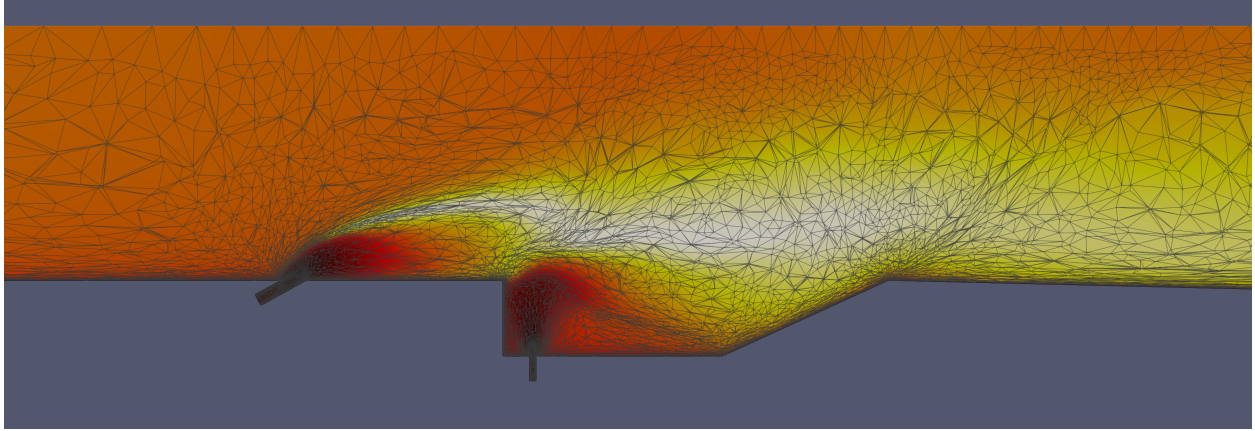


Figure 13: Overview of a Sketch-2-Solution case for chemically reacting flow fed by two hydrogen fuel injectors.

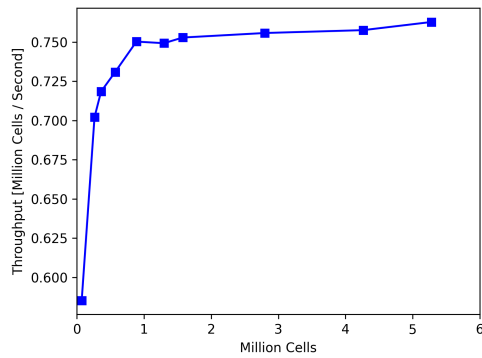


Figure 14: GPU throughput of over 0.75 million cells per second up to 5 million cells.

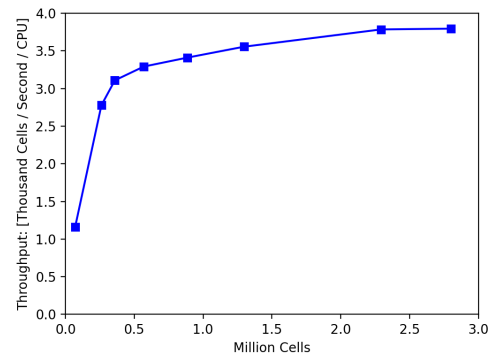


Figure 15: CPU throughput of nearly 4 thousand cells per second per CPU core up to 3 million cells.

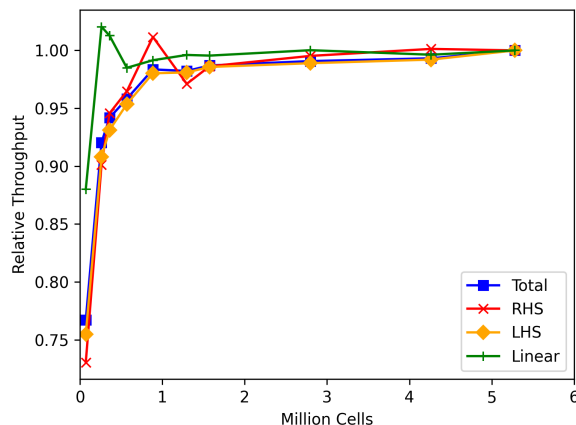


Figure 16: Normalized throughput of different phases of the solver for a 7 species reacting case.

the prototype solver in most cases is either a parallel loop over either the cells or the faces in the mesh. Therefore, it is not surprising that both the single- and multispecies cases reached peak throughput at comparable grid sizes since ideal throughput on a GPU is reached by exposing enough parallel tasks.

GPU Strong Scaling Study The 5.3 million cell grid case was also used to conduct a strong scaling study. The grid was run on 1, 2, 4, and 8 A100 80 GB GPUs. The speedup results are shown in Figure 17. Similar to the single species case GPU speedup study, we see that the LHS exhibits the best speedup of any phase of the solver due to requiring a large amount of compute resources and requiring no communication. Both the linear solve and RHS phases exhibit an overall slowdown on 8 GPUs. However, since the linear solve and the RHS are both a smaller fraction of the total iteration cost, we continue to see benefits overall all the way to 8 GPUs, although with significantly reduced efficiency. Note that a 5.3 million cell grid, split into 8 partitions, yields roughly only 660 thousand cells per GPU, and as shown in Figure 16 this is below the minimum grid size needed to obtain peak throughput on a single GPU. As a practical matter, this simply means that with the current prototype solver, a 5.3 million cell grid is simply too small to efficiently run on so many GPUs, even before any communication cost is considered.

The relatively low memory available on a GPU poses a challenge for multispecies cases. The single GPU throughput measurements from the previous section indicate that a single GPU matches the throughput of nearly 200 CPU cores; however, it is typical for CPU-based High Performance Computing (HPC) nodes to be designed with at least 1 GB of system memory per core. For example, the CPU nodes used for testing in this work have, at minimum, 96 GB, of system memory shared with 40 CPU cores (nearly 2.5 GB of memory per core on average). However, a single A100 GPU has a maximum of 80 GB of GPU memory, while a comparable 200 CPU core job would likely have access to 400 GB of memory. This result highlights a narrow window for single GPU strong scaling, especially for multispecies cases. To achieve near ideal throughput the grid partition assigned to a single GPU must be sufficiently large, but if the grid partition is too large the solver may not fit into GPU memory. Some strategies, such as decoupled implicit chemistry [24] or matrix free nonlinear solvers [26], could be leveraged to lower the memory requirement of the solver.

GPU Weak Scaling Study The final test is a weak scaling study from 1 to 8 GPUs. Grids sized 6.5, 12.3, 22.8, and 38.5 million cells were run on 1, 2, 4, and 8 A100 80 GB GPUs. Figure 18 shows the throughput numbers along with what an ideal throughput would be for each grid size. We see that the solver scales well to 4 GPUs but suffers noticeably at 8 GPUs.

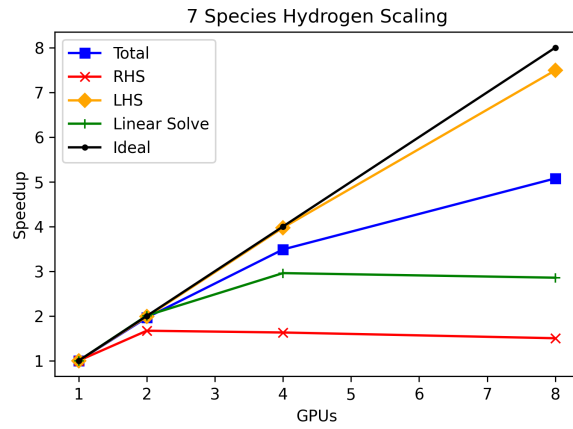


Figure 17: Strong scaling study results of 5.3 million cell grid between 1 and 8 A100 80GB GPUs.

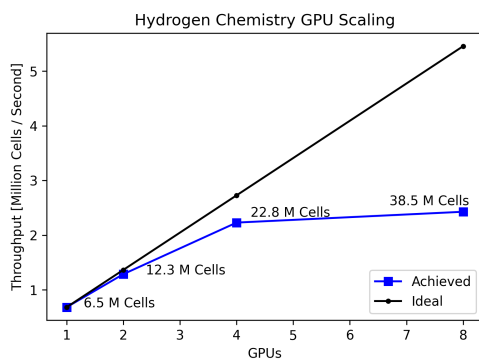


Figure 18: Weak scaling throughput.

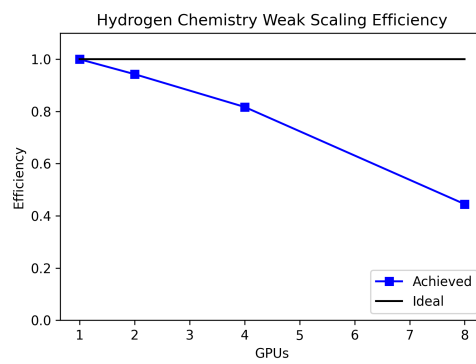


Figure 19: Weak scaling efficiency.

Cross GPU communication is the least mature portion of the prototype solver and continued improvements are planned to address the reduction in efficiency when multiple GPUs are used. The first improvement is to interleave computation and communication as currently the prototype solver does not hide any of the GPU to GPU communication cost behind additional compute. The second planned improvement is to leverage an MPI implementation that is GPU aware. Currently, all the GPU to GPU communication must first be explicitly copied into the host machine RAM before being sent off to MPI. Some MPI implementations that are GPU aware can communicate directly from GPU to GPU without needing this extra copy into the host machine. Using a GPU-aware MPI implementation may help to dramatically reduce the communication cost that drives the inefficiency in the RHS and linear solve phases.

CONCLUSIONS

This work documents the VULCAN-CFD transition to a new C++ and Kokkos implementation; we have unveiled promising preliminary performance results that underscore the potential of this transformative endeavor. This work represents the conclusion of a risk reduction effort into GPU acceleration. Now that all major algorithm patterns have been ported to GPUs the authors are confident that there are no “show stoppers” in porting all the VULCAN-CFD capabilities to take substantial advantage of GPU acceleration.

Our comparison between the CPU performance of the prototype implementation and the original Fortran version reveals that the new C++ implementation nearly mirrors that of the original version, while showing substantial performance benefits when utilizing GPU hardware. This finding is significant in that a single solver written in Kokkos functions, no worse on CPUs (as compared with the “classic” solver), while offering substantial performance boost on GPU architectures. We’ve found that a single NVIDIA A100 GPU is equivalent to the computational power of over 200 CPU cores. This was demonstrated by multiple performance evaluation approaches and using single species calorically perfect gas and multicomponent thermally perfect species reacting cases.

Further continued development is ongoing and is influenced by some of the shortcomings identified during this study. One major component of further development will be the focus on multi-GPU performance. The strong and weak scaling studies have provided valuable insights into the parallel performance of the prototype implementation across multiple GPUs. The current implementation yielded acceptable scalability results up to 4 GPUs, and we acknowledge that there’s a need for further performance enhancements for hardware configurations utilizing 8 (or more) GPUs. Additionally, the prototype implementation of the Jacobian matrix formulation is marginally slower than that of the “classic” implementation. To this point, both multi-GPU support and Jacobian matrix formulation have not received significant developer attention. Most effort has been in porting different portions of the RHS calculation (inviscid and viscous fluxes, as well as chemical and turbulent source terms). The results of this study motivates additional development efforts to further enhance the performance of these two key shortcomings.

The findings in this work serve as a call to action, spurring us to continue our pursuit of optimizing parallel performance to ensure that VULCAN-CFD excels even in the most demanding computational scenarios. These results reaffirm our commitment to preserving the solver’s legacy while embracing modern computing techniques and hardware. With the completion of this risk reduction study, we have confidently concluded that GPUs are a viable and necessary modernization effort. We are continuing to add additional capabilities to the initial prototype implementation in preparation for inclusion with the next release of VULCAN-CFD.

ACKNOWLEDGEMENTS

The authors would like to thank the NASA Hypersonic Technologies Project for their funding and support of this development effort. The authors would also like to thank Chip Jackson from the OVERFLOW development team for his help in configuring MPI with multiple GPUs.

REFERENCES

- [1] Nastac, G., Walden, A., Wang, L., Nielsen, E. J., Liu, Y., Opgenorth, M., Orender, J., and Zubair, M., **A Multi-Architecture Approach for Implicit Computational Fluid Dynamics on Unstructured Grids**, AIAA Paper 2023-1226 (2023).
- [2] Jackson, C. W., Appelhans, D., Derlaga, J. M., and Bunning, P. G., **GPU Implementation of the OVERFLOW CFD Code**, AIAA Paper (2024), in preparation.
- [3] Latt, J., Malaspinas, O., Kontaxakis, D., Parmigiani, A., Lagrava, D., Brogi, F., Belgacem, M. B., Thorimbert, Y., Leclaire, S., Li, S., Marson, F., Lemus, J., Kotsalos, C., Conradin, R., Coreixas, C., Petkantchin, R., Raynaud, F., Beny, J., and Chopard, B., **Palabos: Parallel Lattice Boltzmann Solver**, Computers & Mathematics with Applications, 81:334–350 (2021), development and Application of Open-source Software for Problems with Numerical PDEs.
- [4] Piscaglia, F. and Ghioldi, F., **GPU Acceleration of CFD Simulations in OpenFOAM**, Aerospace, 10(9) (2023).
- [5] ANSYS, **ANSYS GPU Accelerator Capabilities**, <https://www.ansys.com/content/dam/it-solutions/platform-support/gpu-accelerator-capabilities-2021-r1.pdf> (2021).
- [6] Carter Edwards, H., Trott, C. R., and Sunderland, D., **Kokkos: Enabling manycore performance portability through polymorphic memory access patterns**, Journal of Parallel and Distributed Computing, 74(12) (7 2014).
- [7] Herten, A., **GPU Vendor/Programming Model Compatability Table**, <https://x-dev.pages.jsc.fz-juelich.de/2022/11/02/gpu-vendor-model-compat.html> (2023).
- [8] Xue, W., Jackson, C. W., and Roy, C. J., **Multi-CPU/GPU parallelization, optimization and machine learning based autotuning of structured grid CFD codes**, AIAA Paper 2018-0362 (2018).
- [9] Xia, Y., Luo, L., and Luo, H., **OpenACC-based GPU Acceleration of a 3-D Unstructured Discontinuous Galerkin Method**, AIAA Paper 2014-1129 (2014).
- [10] Howard, M., Bradley, A., Bova, S. W., Overfelt, J., Wagnild, R., Dinzi, D., Hoemmen, M., and Klinvex, A., **Towards Performance Portability in a Compressible CFD Code**, AIAA Paper 2017-4407 (2017).
- [11] White, J. A., Baurle, R. A., , Passe, B. J., Spiegel, S., and Nishikawa, H., **Geometrically Flexible and Efficient Flow Analysis of High Speed Vehicles Via Domain Decomposition , Part 1 , Unstructured-grid Solver for High Speed Flows** (2017).
- [12] White, J. A., Nishikawa, H., and O’Connell, M., **F-ANG+: A 3-D augmented-stencil face-averaged nodal-gradient cell-centered finite-volume method for hypersonic flows**, in *SciTech 2022 Forum*, 2022-1848, AIAA, San Diego, CA (2022).
- [13] Zubair, M., Nielsen, E., Luitjens, J., and Hammond, D., **An Optimized Multicolor Point-Implicit Solver for Unstructured Grid Applications on Graphics Processing Units**, in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pages 18–25 (2016).
- [14] Thompson, K. B. and O’Connell, M. D., **Streamlined Convergence Acceleration for CFD Codes**, AIAA Paper 2018-3709 (2018).
- [15] O’Connell, M. D., Whitey, J. A., and Thompson, K. B., **Towards an Automated Unstructured Grid Adaptation Workflow with VULCAN**, AIAA Paper 2021-2701 (2021).
- [16] Thompson, K. B., Nishikawa, H., and Padway, E., **Economical Third-Order Methods for Accurate Surface Heating Predictions on Simplex Element Meshes**, Tech. Rep. 2023-2629 (2023).

- [17] Rumsey, C. L., ***Consistency, Verification, and Validation of Turbulence Models for Reynolds-Averaged Navier—stokes Applications***, Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering, 224(11):1211–1218 (2010).
- [18] Park, M. A., ***Anisotropic Output-Based Adaptation with Tetrahedral Cut Cells for Compressible Flows***, Ph.D. thesis, Massachusetts Institute of Technology (Sep. 2008).
- [19] Kleb, B., Park, M. A., Wood, W. A., Bibb, K. L., Thompson, K. B., Gomez, R. J., III, and Tesch, S. H., ***Sketch-to-Solution: An Exploration of Viscous CFD with Automatic Grids***, AIAA Paper 2019–2948 (2019).
- [20] NVIDIA, ***NVIDIA A100 Specifications***, <https://www.nvidia.com/en-us/data-center/a100/> (2023).
- [21] Intel, ***Intel Xeon Gold 6148 Processor Specifications***, <https://www.intel.com/content/www/us/en/products/sku/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz/specifications.html> (2023).
- [22] Spalart, P. R. and Allmaras, S. R., ***A One-Equation Turbulence Model for Aerodynamic Flows***, AIAA Paper 92–439 (1992).
- [23] Haimes, R. and Dannenhoffer, J. F., III, ***The Engineering Sketch Pad: A Solid-Modeling, Feature-Based, Web-Enabled System for Building Parametric Geometry***, AIAA Paper 2013–3073 (2013).
- [24] Nielsen, T. B., ***Large Eddy Simulation of High-Speed Premixed Turbulent Combustion***, Ph.D. thesis, North Carolina State University (Jul. 2018).
- [25] Brogi, F., Bnà, S., Boga, G., Amati, G., Ongaro, T. E., and Cerminara, M., ***On floating point precision in computational fluid dynamics using OpenFOAM***, <https://arxiv.org/pdf/2209.06105.pdf> (2023).
- [26] Tu, S., Watts, M., Fuller, A., Patel, R., and Aliabadi, S., ***Development and performance of CaMEL_Aero, a truly matrix-free, parallel and vectorized unstructured finite volume solver for compressible flows***, in *Proceedings of the 25th Army Science Conference*, pages 27–30 (2006).

Appendix A: Current GPU Features

Inviscid Fluxes	Status
LDFSS	Completed
Roe	Completed
HLLE++	New Feature
Low Dissipation Roe	New Feature
Rusanov	New Feature
HLLC	Planned Near Term
Inviscid Discretization Schemes	Status
Fromme's 2nd order scheme	Completed
Nishikawa's 3rd order scheme	New Feature
Nishikawa's LP-UMUSCL	Planned Far Term
Viscous Discretization Schemes	Status
Face-Tangent	Completed
Alpha-Damping	Completed
Gradient Options	Status
Cell Based	Completed
C-ANG	Completed
F-ANG	In Progress
Green-Gauss	Not Planned
Linear Least Squares	Completed
Quadratic Least Squares	Completed
Automatic stencil augmentation	Completed (up to degree 3 neighbors)
Slope Limiters	Status
Venkatakrishnan	Completed
Minmod	Completed
van Albada	Completed
Koren	Not Planned
Nishikawa-6	Completed
Face Based Limiting	Completed
Node Based Limiting (MLP)	Completed
Gnoffo and White Pressure	Partially Completed (missing White expansion term)
Turbulence Modeling	Status
Laminar	Completed
Spallart-Allmaras	Completed
Spallart-Allmaras-NEG	Completed
Spallart-Allmaras-Catris	New Feature
Spallart-Allmaras-QCR-2000	Completed
Spallart-Allmaras-QCR-2013	Completed
Spallart-Allmaras-QCR-2013-V	Completed
Spallart-Allmaras-QCR-2020	In Progress
Spallart-Allmaras RC	In Progress
Spallart-Allmaras noft2	Completed
Menter-Baseline-1994	Planned
Menter-Baseline-2003	Planned
Menter-SST-2003	In Progress
Wilcox 1998	Planned

Wilcox 2006	In Progress
RG-EASM-KO-2003	Not Planned
Thivet Realizability	Planned
Durbin Realizability	Planned
Wilcox wall functions	Planned

Chemical Modeling	Status
Chemically frozen	Completed
finite rate w/ Arrhenius kinetics	Completed
finite rate w/ Arrhenius and pressure dependent kinetics	Planned Near Term
Non-integer Stoichiometric coefficients	Planned Near Term

Transport Modeling	Status
Collision cross sections	Completed
species transport via Sutherland's Law	Planned Near Term
mixture transport via Mason and Saxena	Planned Near Term

Boundary Conditions	Status
Extrapolation	Completed
Extrapolation 2nd order	Completed
Mixed Outflow	Completed
Pressure Outflow	Not Planned (superceded by Mixed Outflow)
Fixed Inflow	Completed
Reference Inflow	Completed
Total Pressure, Total Temperature Inflow	Completed
Characteristic Inflow / Outflow	Completed
Isothermal Wall	Completed
Adiabatic Wall	Completed
Radiative Equilibrium Wall	Completed
Tangent Flow	Completed
Slip Wall	Completed
Blowing Wall (Ablation)	Planned Far Term
Velocity, Static Temperature Inflow	In Progress
Symmetric Flow	Planned Near Term
Periodic Flow	Planned Far Term
Profile Condition	Planned
Thermally Coupled Wall	Planned
Wall function variants	Planned
Mass Flux Outflow	Planned
Mass Flux, Total Temperature Inflow	Planned
Bleed variants	Planned

Miscellaneous	Status
2D unstructured	Completed
arbitrary polyhedra	Planned Far Term
Jacobian-Free Newton Krylov	Completed
Defect-Correction Iteration	In Progress
Automated CFL	Completed
time history specification	In Progress
initialization regions	Completed
ignition regions	Planned
turbulence suppression regions	Planned
decoupled chemistry	Planned Far Term