Porting a Computational Fluid Dynamics Code with AMR to Large-scale GPU Platforms

Joshua H. Davis†, Justin Shafner*, Daniel Nichols†, Nathan Grube*, Pino Martin*, Abhinav Bhatele†

†*Department of Computer Science, University of Maryland*
**Department of Aerospace Engineering, University of Maryland*
College Park, Maryland 20742 USA
E-mail: {jhdavis, jshafner, dnicho, ngrube, mpmartin}@umd.edu, bhatele@cs.umd.edu

***Abstract*—Accurate modeling of turbulent hypersonic flows has tremendous scientific and commercial value, and applies to atmospheric flight, supersonic combustion, materials discovery and climate prediction. In this paper, we describe our experiences in extending the capabilities of and modernizing CRoCCo, an MPI-based, CPU-only compressible computational fluid dynamics code. We extend CRoCCo to support block-structured adaptive mesh refinement using a highly-scalable AMR library, AMReX, and add support for a fully curvilinear solver. We also port the computational kernels in CRoCCo to GPUs to enable scaling on modern exascale systems. We present our techniques for overcoming performance challenges and evaluate the updated code, CRoCCo v2.0, on the Summit system, demonstrating a 6× to 44× speedup over the CPU-only version.**

***Index Terms*—hypersonics, computational fluid dynamics, adaptive mesh refinement, GPU computing**

I. Introduction

The potential for hypersonic flight and commercial space access to significantly enhance life on earth is immeasurable – the growth of new markets, the discovery of new materials, and our ability to monitor climate change are just a few of the many possibilities that hypersonic and space flight will bring. Innovation in hypersonics is guided by physics-based simulation tools. However, a key challenge in these efforts is to overcome the extreme computational demands of high-fidelity (HiFi) modeling for hypersonic systems. Resolving all important spatial and time scales along with highly nonlinear multi-physics interactions, especially at the full vehicle level, goes beyond even the current state-of-the-art.

This is especially difficult in hypersonic applications where the flow around vehicles can span large spatial domains, but dictates resolving of turbulent features on the order of micrometers. To date, the computational cost of including such resolution across an entire domain is impractical. Alternative methods must be developed to selectively reduce grid density while maintaining HiFi capabilities. Overcoming these numeric and computational challenges is more difficult when adapting an existing code. Such scientific software is often optimized for older-generation hardware and may be designed in a manner that does not readily adapt to modern accelerators such as GPGPUs. Additionally, such software may be designed using older programming models and may not integrate well with state-of-the-art frameworks. This issue is particularly relevant for solvers working on curvilinear grids, which are not typically supported by modern solver frameworks.

In this paper, we address these problems by extending an existing computational fluid dynamics (CFD) code called `CRoCCo`. `CRoCCo` is a hypersonic flow simulation with shock capturing and high-bandwidth-resolving efficiency, validated for unsteady, highly-turbulent, high-enthalpy, chemically-reacting hypersonic flows on structured grids. The code can operate in direct numerical simulation (DNS) or turbulence modes. The former solves equations from first principles [1]–[3] and the latter computes large eddy simulations (LES) [4], [5] or Reynolds-averaged Navier-Stokes (RANS) simulations.

To tackle the aforementioned difficulties in efficiently resolving high-fidelity hypersonic numerics we extend the capabilities in `CRoCCo` to support adaptive mesh refinement (AMR) for a fully curvilinear solver. AMR dynamically distributes grid densities allowing for selective levels of refinement to change and adapt with relevant flow features. This technique is essential to reducing the time-to-solution of CFD simulations on massive computational domains including those required for current flight vehicles such as the BoLT hypersonic demonstrator [6] or Mars supersonic retropropulsion (SRP) configurations [7]. We further extend `CRoCCo` to employ GPU accelerators to reduce time-to-solution and maximize utilization of cutting-edge computing platforms.

We implement the AMR and GPU capabilities by adapting our existing numerics to efficiently use high-level paradigms supplied by the AMReX [8] framework. AMReX provides abstractions for grid representation, domain decomposition, communication, and GPU acceleration with load balancing and mesh refinement capabilities. While AMReX is a state-of-the-art framework for CFD simulations, it does not support curvilinear grids. Additionally, when accelerated with GPUs, it does not trivially scale to large node counts.

We evaluate the performance of different versions of `CRoCCo` on the Summit system at Oak Ridge National Laboratory (ORNL). This work makes the following major contributions:

- We demonstrate a novel methodology for enabling adaptive mesh refinement with AMReX in a curvilinear CFD solver, specifically adapting the handling of grid metrics, interpolation, and regridding.

- We describe and provide insights from our experiences converting complex high-fidelity numerics in Fortran to GPU-enabled C++.
- We present kernel-level performance as well as strong and weak scaling performance and profiling results for the augmented GPU- and AMR-enabled CFD code on Summit at ORNL.
- We demonstrate up to 44× speedup in the GPU-enabled code over the previous CPU implementation.

II. Background

In this section, we provide background on the numerics implemented in `CRoCCo` as well as the technique of adaptive mesh refinement (AMR) as used in high-fidelity modeling.

A. High-fidelity simulation numerics for hypersonics

`CRoCCo` solves the conservative form of the equations governing fluid motion, namely the conservation of the species mass, momentum, and total energy equations of the form:

$$\begin{aligned}\frac{\partial \rho_s}{\partial t} + \frac{\partial}{\partial x_j}\Big(\rho_s u_j + \rho_s v_{sj}\Big) &= w_s \\ \frac{\partial \rho u_i}{\partial t} + \frac{\partial}{\partial x_j}\Big(\rho u_i u_j + p\delta_{ij} - \tau_{ij}\Big) &= 0 \\ \frac{\partial E}{\partial t} + \frac{\partial}{\partial x_j}\Big((E+p)u_j - u_i\tau_{ij} + q_j + \sum_s \rho_s v_{sj} h_s\Big) &= 0\end{aligned} \tag{1}$$

where w_s represents the rate of production of species s due to chemical reactions; ρ_s is the density of species s; u_j is the mass-averaged velocity in the j direction; v_{sj} is the diffusion velocity of s; p is the pressure; τ_{ij} is the shear stress tensor given by a linear stress-strain relationship; q_j is the heat flux due to temperature gradients; h_s is the specific enthalpy of species s; and E is the total energy per unit volume given by

$$E = \sum_s \rho_s c_{vs} T + \frac{1}{2}\rho u_i u_i + \sum_s \rho_s h_s^\circ, \tag{2}$$

where c_{vs} is the specific heat at constant volume of species s; and h_s° represents the formation heat of species s. `CRoCCo` solves these equations on generalized curvilinear grids, meaning the problem's physical coordinate systems can be curved and non-uniformly spaced. This physical domain is mapped to a rectangular block-structured computational domain.

We use a finite-difference, weighted essentially non-oscillatory (WENO) method to solve convective fluxes. Our WENO is bandwidth-optimized (WENO-SYMBO) to accurately resolve the smallest-scale features of turbulent flows on a reduced number of grid points. WENO-SYMBO considers multiple stencils around the interface to reconstruct the flux at an interface in each direction. WENO weighs these candidate stencil via local relative smoothness coefficient to apply optimal stencil coefficients to reconstruct the flux. Martín et al. [9] provide further details of a bandwidth-optimized WENO method. We use a 4th-order-accurate, central-difference scheme to compute the viscous fluxes and propagate viscous and convective fluxes in time using a third-order accurate, low-storage Runge-Kutta (RK3) method [10].

`CRoCCo` has been validated against experimental data for direct numerical simulations (DNS) [3], [11]–[13]. `CRoCCo` also can resolve hypersonic turbulent flows using large eddy simulation (LES) techniques which filters and does not resolve on the grid the highest frequency energy content. The technique allows for a 90% reduction in grid size relative to DNS and thus faster times to solution. For LES, `CRoCCo` solves the filtered form of Equation 1, which includes subgrid scale (SGS) models that have also been validated against experimental and DNS data for hypersonic turbulent flows [4], [5]. The code was originally written in Fortran with MPI, and this version remains extremely efficient on massively-parallel CPU-based supercomputers.

B. Adaptive mesh refinement in the context of HiFi modeling

Adaptive mesh refinement (AMR) allows for computational domains to be locally and dynamically coarsened or refined during a simulation. In practice, AMR results in an overall decrease in time-to-solution by reducing the number of grid points in regions lacking fine flow scales. Block-structured AMR (BS-AMR) is a series of logically rectangular grids, or 'patches'. Unlike a hierarchical quadtree or octree AMR framework, there is no parent-child relationship between patches of differing levels of refinement. Instead, the patches are overset as presented in Figure 1. Hereafter, we mean "AMR" to refer specifically to block-structured AMR, which `CRoCCo` exclusively employs.

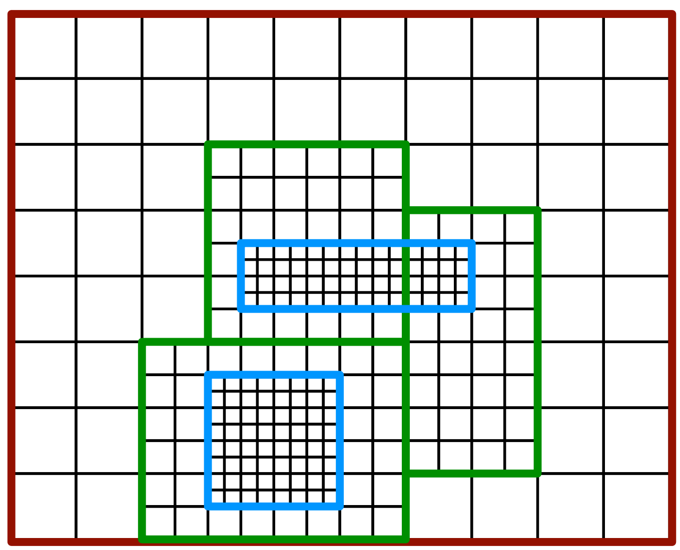

Fig. 1. Example of a 2D AMR grid, showing three levels in total. The coarsest grid remains active across the entire domain, while the finer and finest patches are overset as contiguous block-structures.

In AMR, there are three main factors to consider: (1) how frequently should a patch check whether to refine or to coarsen, (2) the criteria to refine/coarsen patches, and (3) how to exchange information between patches of different levels, or 'interface'. Local physics, numerical ghost-point requirements, and solver order-of-accuracy or bandwidth properties all influence the optimal implementation of these factors. Unique approaches to regrid criteria and interpolation required for `CRoCCo` will be discussed in Section III-C.

AMR regridding frequency: The Courant-Friedrichs-Lewy condition (CFL) is a numerical stability parameter to restrict the number of grid points that a fluid element can travel in a simulation time step. The simplest formulation for the CFL in a one-dimensional flow is:

$$\mathrm{CFL} = \frac{(|u| + a)\Delta t}{\Delta x} \leq 1 \tag{3}$$

where Δt is the time step, Δx is the grid spacing, u is the velocity of the fluid, and a is the local speed of sound. The desired stable CFL number is uniquely supplied for stability of the numerical method. For the RK3 method in `CRoCCo`, CFL≤ 1. To accurately track features using AMR, the application estimates the optimal regridding frequency as the number of timesteps it takes for information to travel from the center of a patch to the nearest fine/coarse interface. Since these interfaces require numerical interpolation, the regrid frequency must be high enough so that flow features do not convect across different level patches.

AMR regridding criteria: For compressible hypersonic problems, AMR is most useful for refining near unsteady, turbulent flow structures and therefore the regridding criteria should seek to capture such effects. Often the most challenging and interesting regions of flow are near shocks, or regions of discontinuous fluid density, ρ. Criteria based on the local gradients of density, $\nabla\rho$, or the local gradient of momentum $\nabla(\rho u_i)$ can aid solvers in capturing shock structures.

AMR fine/coarse interpolation: Determining values at fine ghost points requires interpolation across interfaces. AMReX provides a trilinear interpolator that interpolates fine information by considering information from eight neighboring coarse vertices of a tetrahedral cell. Interpolation introduces error by way of numerical dissipation. Selecting an interpolation technique that is similar in formulation to the chosen numerical method can prevent error and noise propagation [14].

III. Developing Curvilinear AMR Capabilities in CRoCCo

Below, we describe our approach to designing and implementing AMR support in `CRoCCo`, and designing a unique approach to support curvilinear coordinates via AMReX.

A. Enabling AMR in CRoCCo

`CRoCCo` follows a typical time-marching scheme. Algorithm 1 outlines the code for the main routine, including the regridding step (`Regrid`) before calling the `RK3` function. Algorithm 2 describes the implementation of the `RK3` function. We distinguish between `FillPatch`, we adapt from AMReX's `Advection_AmrCore` example, and `BC_Fill`, a custom written kernel. The former handles the main ghost exchange between patches, including performing the interpolation across fine/coarse AMR interfaces. The latter `BC_Fill` applies the physical boundary conditions of the problem. `AverageDown` sets covered coarse AMR cells to be equal to the average of the covering fine cells.

Algorithm 1 The main loop in CRoCCo

```
InitGrid()
InitGridMetrics()
InitFlow()

for n = nstart, nend do
    if mod(step, regridFreq) == 0 or step == 0 then
        Regrid()
    end if
    ComputeDt()
    RK3()
end for

Finalize()
```

Algorithm 2 RK3 Advance Stage in CRoCCo

```
for RKstage = 1, 3 do
    for n = 0, nlevels do
        FillPatch()
        BC_Fill()
        WENOx()
        WENOy()
        WENOz()
        Viscous()
        Update()
        if RKStage == 3 then
            AverageDown()
        end if
    end for
end for
```

B. Implementing AMR using AMReX

We adapt block-structured AMR into `CRoCCo` using the AMReX library [8] which allows for near-seamless integration of Fortran numerical kernels into a Cartesian AMR framework. This allows for the testing of integrating AMR into the numerical methods without the development costs of creating AMR and load balancing algorithms from scratch.

The AMReX framework divides the domain into rectangular patches of grid points, which can have varying degrees of refinement, in order to implement adaptive mesh refinement (AMR). Patches at more refined levels represent the corresponding physical region with a larger number of grid points, while patches at more coarse levels represent the region with a smaller number of grid points. How AMReX carries out this decomposition can be controlled using various input deck parameters, including the number of points in each direction in the domain (at the coarsest level) and the blocking factor. The size of the particular problem determines the number of points in each direction, while the blocking factor, a per-direction setting, must be at least the number of ghost points required in our numerics in each direction, so we set it to 8. Here, we mean "ghost points" to refer to cells beyond the bounds of a patch that must be retrieved from other patches.

Implementing the large number of ghost points and constraints of `CRoCCo` numerical methods into grid refining and message passing was straightforward due to the simple nature of AMReX source kernels. AMReX also allows for the addition of custom interpolators, time integrators, and refinement routines that allowed our team to develop curvilinear coordinate AMR despite the library's intended use on Cartesian grids. In addition, the library and underlying data structures built into AMReX are GPU-portable and architecture independent that reduced the time of the development cycle to offload `CRoCCo` to GPUs. Other AMR frameworks will be discussed in Section VII-B.

The `Fill_Patch` method handles ghost exchange between patches, as described in Section III-A. The `FillPatch` method employs two functions provided by AMReX's `FillPatchUtil` to fill ghost cells: `FillPatchSingleLevel` for patches at the coarsest level, and `FillPatchTwoLevels` for patches at finer levels. `FillPatch` also passes our custom WENO interpolation method to be described in Section III-C into `FillPatchTwoLevels`. AMReX carries out load balancing of the patches across MPI ranks at each level independently, in sequence. The default load balancing algorithm, which we use, is a space-filling Z-Morton curve. Since AMReX has demonstrated excellent scaling results on modern GPU-based supercomputers, including Summit, with similar workloads to `CRoCCo`, we are confident in relying on their provided parallelization and load balancing methods to interface with MPI [8], [15].

The `FillPatchSingleLevel` function only involves point-to-point MPI communication between neighboring patches to receive ghost points. There are two additional parallel communication calls within the code that involve global communication calls. The first is found within the `ComputeDt` routine which checks each grid point's solution to determine the most effective next timestep that adheres to the CFL criteria introduced in Section 3. Since every patch must iterate using the same timestep, dt, which must be computed using the global minimum derivative of time, there is a global `MPI_Reduce` call using an AMReX wrapper function `ReduceRealMin(dt)`.

The second global communication call is an AMReX function `ParallelCopy` in our custom curvilinear interpolator, which is called from `FillPatchTwoLevels` in `FillPatch`. This copies the coordinates from the main grid `MultiFab` to a temporary one with additional ghost points to perform curvilinear interpolation.

C. Enabling curvilinear coordinates in AMR

`CRoCCo` has always supported curvilinear grids as they are necessary for gathering HiFi datasets of hypersonic flows around compression corners, re-entry vehicles, and other complex geometries. Our numerical kernels contain full curvilinear support, regardless of the grid type. However, most AMR libraries (including AMReX) natively support Cartesian grids and cylindrical coordinate transforms, so we undertook a large development effort to adapt the AMReX framework itself to handle the demanding needs of a fully curvilinear solver.

Data management: Curvilinear solvers require more data to solve the governing equations. The non-linear mapping of the physical domain x, y, z onto the computational block coordinates i, j, k makes it necessary to track grid metrics. Both the Cartesian and curvilinear implementations use a type `amrex::MultiFab` to store the primitive variables used in the compressible Navier-Stokes equations. An additional `amrex::MultiFab` stores the five components of the conservative update variable dU at each grid point. For a uniform Cartesian solver, the grid spacing is constant and thus an inexpensive analytical mapping function can be used to pull coordinates such as $x(i) = (i-1) * dx$ making it not necessary to store grid coordinates in large arrays. Curvilinear grids are often generated using combinations of complex hyperbolic and trigonometric functions that justify storing coordinates rather than calculating them on the fly. Therefore the curvilinear code adds a three-component coordinates `amrex::MultiFab`. In addition, solving the curvilinear metrics requires first and second order grid metrics. These are the high-order reconstructions of the first and second derivatives of each i, j, k with respect to x, y, z. In total, we use a 27-component `amrex::MultiFab` to store the metrics. The end result is roughly a three-fold increase in memory usage per core or GPU for the curvilinear code.

Interpolation: The default trilinear interpolation method in AMReX considers the solution at eight vertices surrounding a fine point on a uniform Cartesian grid (see Sec. II-B). Under the constraint of uniform Cartesian coordinates, fine points are always physically located halfway between coarse points and therefore the interpolation coefficients are always a multiple of $1/2$. To adapt this to generalized curvilinear grids, bringing physical coordinates and local grid metrics into the optimized interpolation computations required extra data management. Our custom interpolation scheme accurately weighs interpolation coefficients by spacing in physical curvilinear space. The end result has been sufficient for the double-Mach reflection case tested in this work in Sec. VI, but lacks conservation of quantities across interfaces.

As we move to hypersonic turbulent flows, a higher-fidelity interpolation method will be needed to guarantee conservation across interfaces and mitigate spurious noise introduced across the fine/coarse boundary. As mentioned in Section II-B, we are developing a high-order, bandwidth optimized WENO interpolation scheme, nearly identical to the method Martín et al. use to reconstruct convective fluxes [9]. As the dissipation and order-of-accuracy introduced by the interpolation method will be the same as the numerical methods being employed, there will theoretically be minimal error introduced by a WENO-SYMBO conservative interpolation method.

Regridding: For regridding refinement criteria, `CRoCCo`'s

WENO-SYMBO numerics do not require high grid density near shocks. Thus, CRoCCo includes the option to depart from momentum-based refining criteria (see Sec. II-B) and use AMR exclusively as a turbulence resolving tool.

In terms of implementation, the need to store grid coordinates in memory for curvilinear AMR imposes a set of challenges at large problem scales. The first issue is that when regridding dynamically in AMR, new portions of the domain will be initialized and need their respective coordinates. The first implementation of the code included file I/O at each regrid. A newly formed AMR patch would serially read from a binary file using `std::iostream`. On CPU this added noticeable overhead. For GPU, this approach would be expected to add even more overhead since I/O routines would need to stage data in CPU memory and then copy grid coordinates back to GPU memory. The current implementation allows for the option to read the entire AMR grid into a stored variable. As regrid creates new patches, a simple `getCoords()` call retrieves the data from memory instead of a binary file.

IV. Experiences Porting to Modern Hardware

Preparing the numerics kernels in CRoCCo for modern exascale systems with GPUs required a two-step process – converting Fortran kernels to C++, and then porting the C++ CPU kernels to GPUs. We describe these steps below.

A. Converting Fortran kernels to C++

CRoCCo's core numerics kernels, `WENOx`, `WENOy`, `WENOz`, and `Viscous` (see Algorithm 2), were originally written and optimized for CPU-only systems in Fortran. A C++ AMReX interface with the rest of CRoCCo called these sequential Fortran kernels. We decided to convert these Fortran kernels to C++ in order to maximize compatibility with our existing AMReX-based C++ code. This conversion process required addressing some challenges to preserve readability and the general structure of the code, specifically regarding zero-based indexing and array slicing syntax. During this conversion process, we were able to identify and carry out a number of minor optimizations, with varying impact. Cumulatively, these optimizations help to minimize the observed 1.2x slowdown of the C++ kernels over the Fortran kernels on the IBM POWER9 CPU, as will be presented in Section VI-A.

A particular concern in the translation of our complex numerics kernels from Fortran to C++ was ensuring that no significant floating-point accuracy was lost. Even with apparently-identical code, the differences in information exposed to the compiler in each language lead to different optimizations and order-of-execution, and therefore different floating point results. To assess this, we compared the results of both the C++ and Fortran via the L2-norm of the difference in each flow variable of interest. This value plateaued at $1E-7$ for velocity, density and temperature, which is within machine precision differences given the quantity of operations required to solve for each flow variable.

B. Porting to GPUs

The addition of GPUs to CRoCCo with AMReX adds another level of parallelism below MPI. We compute all major computational steps on the GPU, including the WENO kernels in x, y, and z directions, the `Viscous` kernel, the RK `Update` kernel, the interpolation across interfaces in `FillPatch`, and the time step estimation `ComputeDt` at the beginning of each RK3 stage. All of these kernels contain triple-nested for loops over the three dimensions of a patch, and we parallelize them in each of these dimensions.

We aimed to iteratively offload kernels to the GPU, in order to make development and debugging simpler. We initially explored the use of OpenMP 4.0 directives. However, as development progressed on converting CRoCCo's core structure to AMReX, the built-in GPU support provided by the AMReX framework became a more appealing option. We encountered difficulty in copying complex custom data types to the GPU with OpenMP, particularly with C++ standard library types, as well as some poor initial performance results. For these reasons, we decided to focus on porting to GPUs using the AMReX GPU API, while planning to explore the performance of other GPU programming models in the future. Therefore, we implement both our inter-node parallelism (described in Sec. III-B) and intra-node GPU parallelism using the AMReX framework's communication and kernel launch functions, which completely handle our interfacing with MPI and CUDA.

In porting our kernels to the GPU, we primarily rely on the `amrex::launch` function, which allows portability of user-defined functions between CPU and GPU. In some cases, this required replacing one- and two-dimensional local arrays used for intermediate result storage between outer loop iterations with three-dimensional arrays to avoid data races, as the AMReX GPU API automatically parallelized the kernels in all three dimensions. Additionally, to avoid dynamic memory allocation inside the GPU kernel, which is a major performance impediment, we allocated all of these arrays in GPU global memory from the host code, before kernel launch.

However, this solution was not sufficient for a handful of more complex loop patterns in the kernels. Our higher-order kernels depend on computing large stencils in a few key loops. When we ported these loops to the GPU directly using the `amrex::launch` function, data races resulted, since the threads would write to the same indexes of the scratch arrays in global memory. To resolve this, we moved these more complex stencil loops into dedicated GPU kernels using the `amrex::ParallelFor` function, allowing one thread to compute exactly one cell of the array, including the exterior ghost points needed to provide a complex stencil for each interior cell. Later kernels are therefore able to read in the needed array stencils from global memory without issue. Finally, we employed a reduction pattern using the `amrex::ReduceData` type for computing the explicit time-stepping according to the CFL constraint (see Section II-B).

C. Maintaining correctness

Throughout development our team relied on regular validation runs to ensure code changes did not impact correctness. This procedure was extremely valuable for identifying and triaging several correctness bugs. The shifting of memory buffers and re-structuring of numerics led to large layout and factoring changes in the complex numerical kernels. We thoroughly tested the correctness of these routines by comparing the same L2-norm of the difference in each flow variable. We observed no change in accuracy when running on GPUs and maintained the accuracy reported in Section IV-A.

V. Experimental Setup

In this section, we describe the platform, test case problem, and problem sizes used for the performance evaluation.

A. Platform used

For all runs, we use the Summit supercomputer at Oak Ridge National Laboratory (ORNL). Summit nodes have six NVIDIA V100 GPUs and two 22-core IBM POWER9 CPUs, and the nodes are interconnected using a fat-tree network.

B. The double Mach reflection test case

Our test is the double Mach reflection (DMR) case of Woodward and Colella [16], which has been used extensively in the literature to assess numerical schemes and procedures. The flow is an unsteady planar Mach 10 shock incident on a 30° inviscid compression ramp, depicted in Figure 2. It features a moving shockwave and regions of both turbulence and freestream flow. These are important test elements as they are critical features of hypersonic flows. This case does not require additional elements such as unsteady inflow data from an auxiliary simulation. The problems is statistically homogeneous but locally unsteady along the span. We do not fix the spanwise dimension when scaling calculations. Although unnecessary for this problem, we use general curvilinear coordinates to assess and report AMR savings. The DMR case is easy to run and verify and serves as a representative surrogate including key components of flow environments in hypersonic science applications. We solve the flow in 3D.

C. Weak and strong scaling setup

We use version numbers for easy reference to the different stages of improvements. `CRoCCo` 1.0 refers to the version with C++ AMReX framework and Fortran numerics kernels, with AMR disabled and no GPU support. We obtain `CRoCCo` 1.1 by swapping out the Fortran kernels in 1.0 for new C++ kernels. We obtain `CRoCCo` 1.2 by enabling AMR in 1.1. We obtain `CRoCCo` 2.0 by adding GPU support to 1.2. Note also that we do not present results for the GPU version of `CRoCCo` with AMR disabled, as the non-AMR cases will not fit into the GPU memory of the Summit NVIDIA V100s if the number of nodes is not adjusted.

Section VI presents strong and weak scaling results. In strong scaling mode, we run code versions 1.1, 1.2, and 2.0 on 16 to 1024 nodes with 1.27×10^9 grid points. In weak scaling mode, we run the cases listed in Table I. For AMR, the number of active grid points is dynamic and dependent on time step and flow unsteadiness. Here, we report AMR grid sizes that are the same as those for the AMR-disabled `CRoCCo` v1.1. AMR demonstrates a 89-94% reduction in actual grid points relative to the AMR-disabled solution.

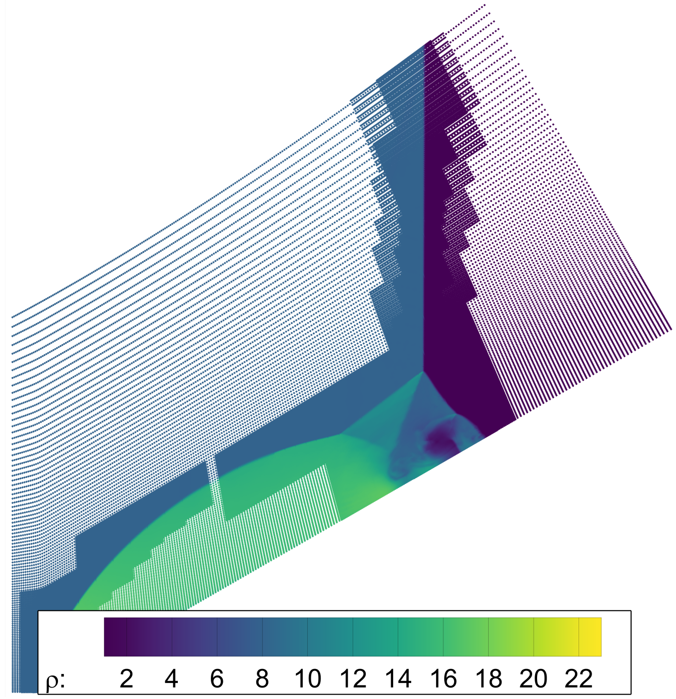

Fig. 2. Density contour of the canonical DMR problem from `CRoCCo` code using three-level curvilinear AMR.

TABLE I
Weak scaling configurations used for evaluating performance

Code Versions	# of Nodes	# of GPUs	# of equivalent grid points
1.1, 1.2, 2.0	4	24	1.64E8
1.1, 1.2, 2.0	16	96	6.55E8
1.1, 1.2, 2.0	36	216	1.47E9
1.1, 1.2, 2.0	64	384	2.62E9
1.1, 1.2, 2.0	100	600	4.10E9
1.1, 1.2, 2.0	256	1536	1.05E10
1.1, 1.2, 2.0	400	2400	1.64E10
1.1, 1.2, 2.0	1024	6144	4.19E10

For accurate grid size scaling, we alter the refinement scheme so that grid size scales with the equivalent problem size. For all scaling experiments we ran 100 iterations and present average walltime per iteration for the last 80 iterations, removing the input from the initial, slower iterations. Data for plots in Sec VI are from single runs, but in practice we observe low variability between runs of the same configuration.

A physical grid aspect ratio of $2:1$ in x and z constrains the DMR problem. Accuracy is independent of y resolution, thus we arbitrarily choose y grid spacing to target grid size

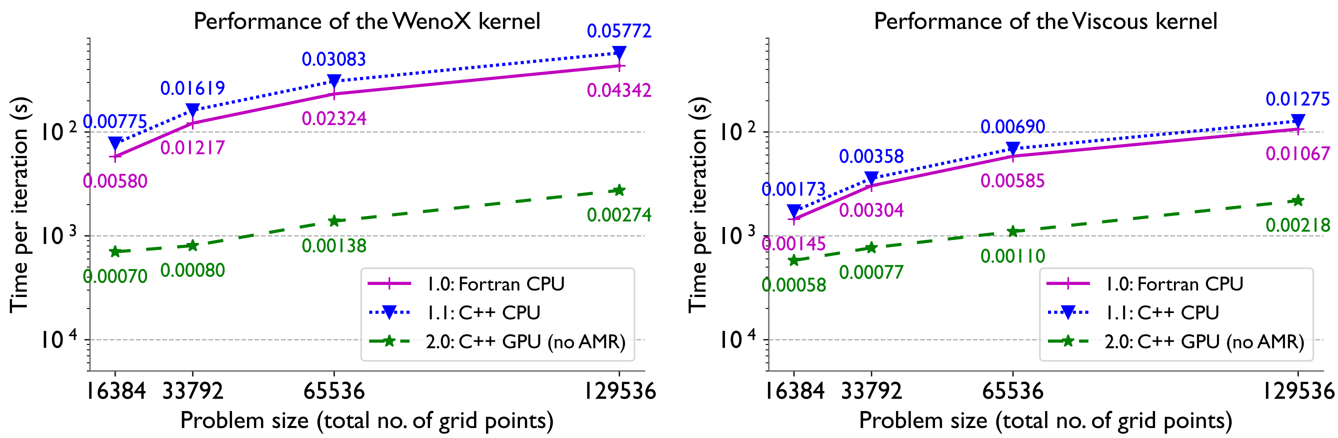

Fig. 3. Comparison of time spent in the WenoX (left) and Viscous (right) kernels per iteration between Fortran on CPU, C++ on CPU, and C++ offloaded to GPU on one 22-core IBM POWER9 processor and one NVIDIA V100 GPU on Summit.

scaling. Through all experiments, the resolution was sufficient such that the load balancer and decomposition remained 3D. To ensure proper gridding, the number in each direction must be divisible by the input blocking factor, which we set to four. Table I lists the weak and strong scaling cases we run, in terms of number of nodes and grid points.

In testing, we found that grid point counts beyond $2.0E5$ spilled out of the 16GB of memory available on Summit V100s. Therefore, we selected the strong scaling problem size to approach this limit for the smallest 16-node case. We determined the weak scaling problem using a similar approach, aiming to maximize GPU utilization without exceeding the available memory. To ensure the large runs would not exceed GPU memory, we set the target number of grid points per GPU at $1.2E5$. The scaling pattern of nodes for weak scaling breaks from perfect doubling (at 4, 36, 100, and 400) to allow for linear problem size scaling while also adhering to the blocking factor and physical $2:1$ point distribution requirements of the AMR and DMR problem physics. We use lightly hand-tuned AMR parameters of 8 for blocking factor in each direction and 128 for maximum grid size in each direction.

VI. Results and Discussion

We first present our performance results of the individual kernels, then present scaling results for the DMR problem, and conclude with performance profiling and analysis.

A. Kernel performance results

To evaluate the performance impact of our Fortran to C++ kernel translation, and our porting of those kernels to the GPU, we measured the time spent in these numerics kernels per `CRoCCo` iteration. Figure 3 presents the time per iteration spent in the `WENOx` and `Viscous` kernels. We scale the kernels over a range of problem sizes run on a single Summit IBM POWER9 22-core processor and NVIDIA V100 GPU, and measure problem size on the x-axis in total coarse grid points in the domain.

First, we observe that the performance impact of our Fortran to C++ kernel translation, described in Section IV-A, is minimal. The C++ versions of the kernels demonstrate a consistent minor slowdown ($\sim$1.2$\times$) over Fortran on the IBM POWER9 CPU architecture. Second, we also observe a very strong performance benefit in porting the kernels from the CPU to the GPU – from a 2.5$\times$ speedup on the smallest problem size for `Viscous` to a 15.8$\times$ speedup on the largest size for `WENOx`, where GPUs are most efficient.

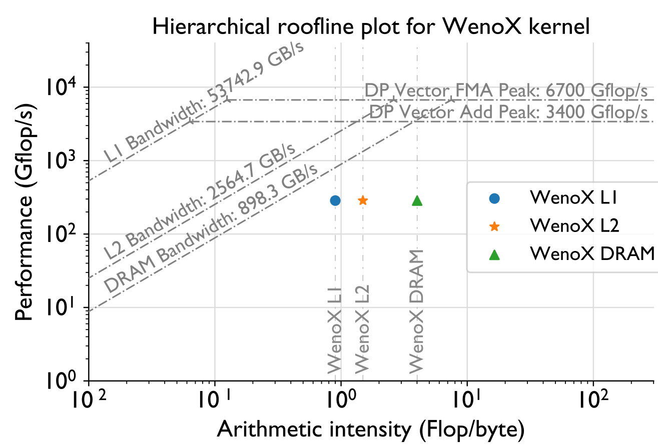

Fig. 4. Hierarchical roofline plot for double-precision flop/s in the WenoX kernel on a Summit NVIDIA V100 GPU.

To more closely examine our GPU kernel performance, we evaluate our key numerical kernels using the roofline model, as described by Yang et al. [17]. Figure 4 presents the results of our roofline evaluation of the `WENOx` kernel, gathered using the NVIDIA Nsight Compute profiler. We choose to omit the roofline plots for the `WENOy`, `WENOz`, and `Viscous` kernels as they are similar to the plot for `WENOx`. All of our numerics kernels, `WENOx`, `WENOy`, `WENOz`, and `Viscous`, achieve about 300 Gflop/s in double-precision (DP). This is approximately 4% of the peak double-precision performance, 7.8 Tflop/s, available on the NVIDIA V100 GPU. The primary

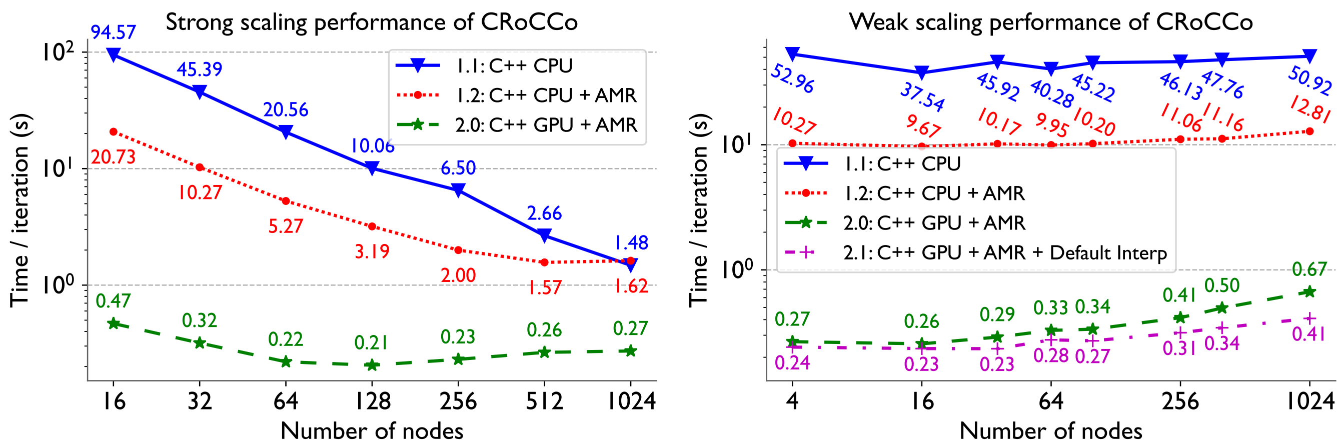

Fig. 5. Strong (left) and weak (right) scaling performance of `CRoCCo` on Summit.

cause for our low achieved flop/s, as revealed by profiling with Nsight Compute, is low theoretical occupancy (12.5%) due to very high register usage arising from the complexity of the physics involved in these high-fidelity kernels. All of our kernels are bandwidth-bound, rather than compute-bound, for L1 cache, L2 cache, and DRAM. Future directions for improving kernel performance include reducing the number of division operations and experimenting with mixed-precision.

B. Scaling performance on the DMR problem

We conducted strong and weak scaling experiments on the three versions of `CRoCCo` discussed in this paper, as described in Section V-C. Figure 5 presents the performance (time per iteration) for strong and weak scaling of `CRoCCo`. In strong scaling of the 1.27E9 grid point case, presented in Figure 5 (left), we observe moderate speedups of the AMR-enabled version 1.2 of `CRoCCo` over the non-AMR version 1.1, and significant speedups of the GPU version 2.0 over the CPU + AMR version 1.2. Speedup from AMR over non-AMR on CPU ranges from 4.6× at the lowest node count to a 1.1× slowdown at the highest, due to the emergence of communication bottlenecks from the AMR procedure. Speedup from GPU over the AMR on CPU case ranges from 44× at the lowest node count to 6× at the highest. Cumulatively, GPU and AMR provide speedups from 201× at the lowest node count to 5.5× at the highest.

We also note that the performance of the GPU version of `CRoCCo` stops improving with additional nodes around 128 nodes in strong scaling, while the CPU version scales well up to 1024. The AMR + CPU version 1.2 also degrades in scaling performance after 512 nodes. Strong scaling an AMR-enabled code on GPUs is associated with a number of challenges, summarized by Katz et al. [15]. Most importantly, due to the high degree of kernel speedup we obtain from moving to the GPU (see Figure 3), the performance bottleneck in `CRoCCo` switches from computation (which is what binds the CPU performance) to communication. Due to global communication in the form of an `amrex::ParallelCopy` in the `FillPatch` routine, our communication costs increase with the number of nodes, which explains why our scaling benefit ends earlier in the GPU version of `CRoCCo`. The end of good scaling behavior at 128 nodes corresponds to when the problem size per GPU becomes too small for the benefit of additional GPUs to outweigh the communication costs of additional nodes. A larger problem size for strong scaling would likely scale further, but would exceed the available GPU memory at lower node counts, massively degrading performance for those runs.

Figure 5 (right) presents weak scaling for the problem sizes set out in Table I. As demonstrated by our strong scaling experiments, the addition of AMR to the CPU version of `CRoCCo` gives moderate speedups while the addition of GPU to the CPU AMR code gives significant speedups. Note that all versions of `CRoCCo` demonstrate slight improvement in walltime per iteration from four to sixteen nodes. This is due to suboptimal load balancing behavior in AMReX at low node counts for certain problem configurations.

We observe in our weak scaling experiments that while the CPU runs demonstrate steady walltime per iteration as node count increases, the GPU version of `CRoCCo` experiences increasing walltime per iteration. Weak scaling efficiency for the GPU + AMR version 2.0 from 4 nodes to 400 nodes is about 54%, and from 4 nodes to 1024 nodes is about 40%. As discussed above, the high degree of speedup obtained by our numerics kernels when moving to GPU results in a strongly communication-bound application, with global communication costs in the `FillPatch` routine increasing with node count.

To quantify the impact of the `ParallelCopy` operation on performance, we present an additional line in Figure 5 (right) in which we swap out our custom curvilinear interpolator scheme (which contains the `ParallelCopy` operation) for AMReX's simpler built-in nodal trilinear interpolator. We name this version `CRoCCo` 2.1. This AMReX trilinear interpolator does not have any global MPI communication. `CRoCCo` 2.1 improves performance and the weak scaling trend of `CRoCCo` 2.0, increasing weak scaling efficiency from 4 nodes to 400 to about 70%. We are exploring options to remove this `ParallelCopy` from our custom curvilinear interpolator, which we expect would enable weak scaling more akin to this improved result. This would primarily entail keeping the entire curvilinear grid in memory from the beginning of the simulation run, eliminating the need for a `ParallelCopy` but at a high memory cost. We finally note that degraded weak scaling behavior in an AMReX-based code when moving to the GPU with high kernel speedup is consistent with results in previous work, including Myers et al. [18] and Katz et al. [15], the former of which also shows degraded scaling efficiency below 60% at large node counts on Summit.

C. Understanding performance of the final implementation

In order to better understand what components of `CRoCCo` other than the curvilinear interpolator are impacting scaling performance, we profile the performance of `CRoCCo` 2.1 (GPU + AMR + Default Interp), presented in Figure 5 (right) as the purple line and discussed in Sec. VI-B. This version lacks the expensive `ParallelCopy` operation which occurs in our custom curvilinear interpolator.

Figure 6 presents the profiling results on Summit for the 2.1 version of `CRoCCo` with this interpolator swap. The profiles depict time spent in each major code region of `CRoCCo` for various node counts. The problem sizes for these experiments follow our weak scaling scheme, described in Table I in Section VI-B. Some smaller sections of the profiles have runtimes too short to be easily visible in the plot. We used the AMReX TinyProfiler tool to collect this data, which provides timer macros to track time spent in code regions. Note that the `Advance` region includes `Fill_BC` physical boundary application as well as the `WENOx`, `WENOy`, `WENOz`, and `Viscous` numerics kernels. The remaining regions correspond to those described in Sec. III-A.

After porting to GPU, `CRoCCo` is strongly communication-bound, as evidenced by the increasing portion of time spent in the `FillPatch` routine. `FillPatch` demonstrates roughly a 40% increase in time spent in `FillPatch` from 4 to 100 nodes, and a 65% increase from 100 nodes to 1024. Conversely, the time spent in `Advance` presented in Figure 6 stays steady as `CRoCCo` scales. This suggests that the GPU kernels are scaling well, and not responsible for the scaling behavior observed in Figure 5 (right).

`Regrid` and `ComputeDt` are the other two regions of `CRoCCo` in which significant MPI communication occurs, and `Regrid` in particular also takes longer as node count increases. `ComputeDt` is a consistently very small portion of runtime, suggesting that the MPI reduction which takes place within it is not a significant bottleneck. While the execution time of the `FillPatch` routine still increases with node count, its scaling is improved compared to what would be observed when using our custom curvilinear interpolator.

Since the remaining performance bottleneck in this faster `CRoCCo` version still lies in `FillPatch`, we present a profile of how `CRoCCo` spends time within `FillPatch` in Figure 7. `ParallelCopy` carries out global communication, while

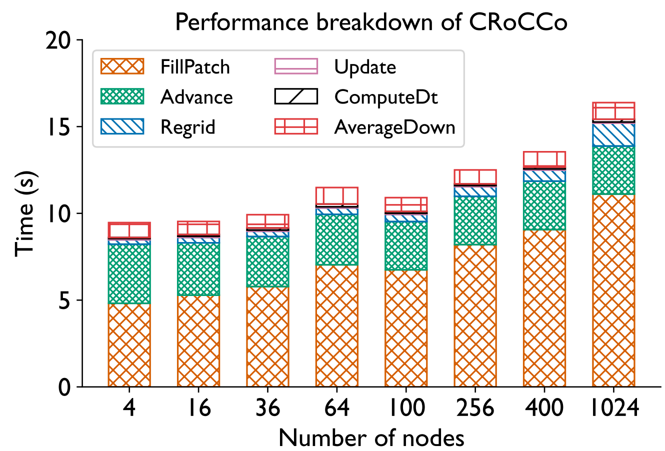

Fig. 6. Decomposition of `CRoCCo` runtime with the default AMReX trilinear interpolator for the weak scaling cases on Summit.

`FillBoundary` carries out boundary cell exchange between neighboring patches using point-to-point MPI messages. The `_finish` and `_nowait` tags in the legend refer to the asynchronous and synchronous versions of each function, respectively. This lower-level profile provides additional insight into the impact of the `ParallelCopy_finish` operation, which increases in execution time as node count goes up.

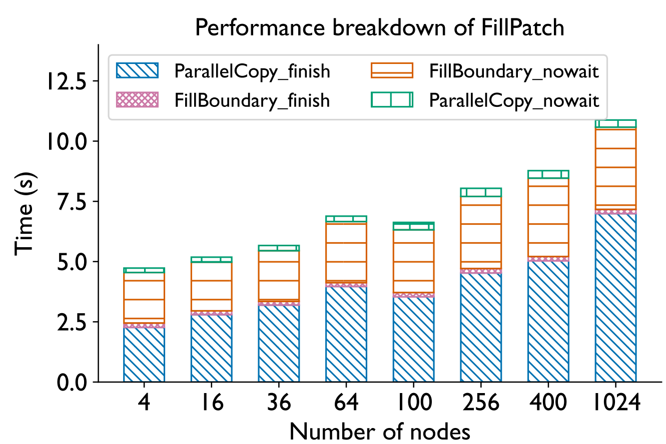

Fig. 7. Decomposition of FillPatch runtime with the default AMReX trilinear interpolator for the weak scaling cases on Summit.

VII. Related Work

In this section we present related work on adaptive mesh refinement on curvilinear grids for high-order methods and porting such applications to exascale software and hardware.

A. Hypersonic flow numerics

Many works have focused on accelerating flow simulations using adaptive mesh refinement and/or GPUs. Atkins and Deiterding [19] present a hypersonic code that makes use of Cartesian AMR with an overset near-body solver to allow for complex geometries. The work of Browne et al. [20] similarly discusses the pairing of high-fidelity modeling with Cartesian AMR coupled to a near-body solver. Both of these approaches avoid curvilinear AMR by creating a fine, unstructured mesh around a complex geometry and interpolating data between it and a coarse Cartesian grid that then performs AMR.

B. Libraries and frameworks for AMR at scale

A number of libraries and frameworks exist which implement adaptive mesh refinement [8], [21], [22]. We selected AMReX for `CRoCCo` for reasons described in Section III-B. Zhang et al. [8] recently introduced AMReX for doing block-structured adaptive mesh refinement. It includes many state-of-the-art methods for exchanging ghost regions, moving computation to GPUs, and grid I/O. SAMRAI [21], [23] and Chombo [22] also both provide structured adaptive mesh refinement. SAMRAI employs the RAJA and Umpire libraries for GPU support and has similar functionality to AMReX. Chombo is an offshoot from AMReX's precursor library, BoxLib [24]. As of its most recent release (3.2.7), it does not have built-in support for GPU offloading [22].

C. Experiences preparing simulations for exascale

Several other works examine preparing particular scientific applications for future exascale systems [15], [18], [25]–[27]. Many of these similarly focus on adding GPU support to fully utilize the hardware on systems like Summit and Perlmutter. They also address parallelization and data handling for their problems that can be scaled to large node counts, focusing on particular algorithmic and engineering decisions pertinent to their domain problem. The most similar of these is Myers et al. [18], who port the WarpX application to Summit using AMReX. WarpX is an electromagnetic particle-in-cell code, and Myers et al. discuss the benefits of optimizing communication, GPU memory usage, and cache utilization for performance on Summit. The WarpX team specifically cites the relative difficulty in scaling the `FillBoundary` routine in AMReX as a the cause for scaling efficiency loss, similar to what we encounter in Sec. VI.

Katz et al. [15] also perform similar work, discussing experiences porting code to GPU systems using AMReX for two applications in the astrophysics domain, Castro and MAESTROeX. Once again, similar to our work, the Castro/MAESTROeX team documents the domination of communication-bound portions of their AMReX-based simulation at large node counts. Mullowney et al. [25] prepare an incompressible-flow simulation, Nalu-Wind, for exascale in the wind energy domain. Kronbichler et al. [27] demonstrate scaling results for a CPU-only version of their lung airflow simulation for the pre-exascale Fugaku system, which resolves the incompressible Navier-Stokes equations at scale. Finally, Bertagna et al. [26] use the Kokkos programming model to scale the nonhydrostatic atmosphere dynamical core of E3SM on Summit, resolving the compressible Navier-Stokes equations.

VIII. Conclusion

We have described our approach and experiences in porting `CRoCCo`, originally a Fortran-based MPI-only code optimized for CPUs, to extreme-scale GPU platforms with the addition of adaptive mesh refinement (AMR) capabilities. Our approach leverages the high-level GPU API offered by AMReX along with careful conversion of Fortran to C++ to productively prepare complex high-fidelity kernels for GPUs. We have also described modifications to the AMReX framework to support a curvilinear flow solver, a previously-unsupported capability. We maintain a high degree of accuracy between the validated Fortran code and the new GPU port, along with speedups ranging from 44$\times$ to 6$\times$ relative to CPU performance.

We offer the following insights from our work:

- When comparing the performance of AMR and non-AMR code versions, ensuring that the refinement of grid points at the finest level for AMR cases is equivalent to the overall refinement of grid points in non-AMR cases allows for a good approximation of relative performance while keeping accuracy as constant as possible.
- Scaling performance is likely to degrade on large node counts for a GPU port of an AMR code due to the relative increase in the ratio of time spent in communication to time spent in computation.
- Parallel copy operations in AMReX can be a communication bottleneck, particularly when global communication is necessary as in curvilinear grids.
- GPU kernels are likely to run with moderate to low utilization of the device when high-fidelity numerics have high register usage, limiting the theoretical occupancy of the streaming multiprocessors.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation Graduate Research Fellowship under Grant No. 1650114. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] I. Beekman, S. Priebe, Y.-C. Kan, and M. P. Martin, "DNS of a large-domain, Mach 3 turbulent boundary layer: turbulence structure," *AIAA 2011-0753*, 2011.

[2] M. P. Martin, "DNS of hypersonic turbulent boundary layers," *AIAA Paper 2004–2337*, 2004.

[3] L. Duan, , I. Beekman, and M. P. Martin, "Direct numerical simulation of hypersonic turbulent boundary layers. Part II: Effect of wall temperature," *J. Fluid Mech.*, vol. 655, pp. 419–445, 2010.

[4] C. Helm and M. Martin, "Large eddy simulations of hypersonic shock-separated flows," *in preparation for submission to Physical Review Fluids*, 2020.

[5] J. Li, S. Priebe, N. Grube, and M. Martin, "Analysis of the large eddy simulation of a shock wave and turbulent boundary layer interaction," *43rd Fluid Dynamics Conference*, 2013.

[6] A. Thome, A. Dwivedi, J. W. Nichols, and G. Candler, "Direct numerical simulation of BOLT hypersonic flight vehicle," *AIAA 2018 Fluid Dynamics Conference*, 2018.

[7] K. Edquist, A. Korzun, A. Dyakonow, J. Studak, D. Kipp, and I. Dupzyk, "Development of supersonic retropropulsion for future Mars entry, descent, and landing systems," *AIAA Scitech 2020*, 2020.

[8] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, 2021. [Online]. Available: https://doi.org/10.1177/10943420211022811

[9] M. Martin, E. Taylor, M. Wu, and V. Weirs, "Bandwidth-optimized weno scheme for the direct numerical simulation of compressible turbulence," *J. Comp. Phys.*, vol. 220, pp. 270–289, 2006.

[10] J. Williamson, "Low-storage runge-kutta schemes," *Journal of Computational Physics*, vol. 35, pp. 48–56, March 1980.

[11] L. Duan, I. Beekman, and M. P. Martin, "Direct numerical simulation of hypersonic turbulent boundary layers. Part III: Effect of Mach number," *J. Fluid Mech.*, vol. 672, pp. 245–267, 2011a.

[12] L. Duan and M. P. Martin, "Direct numerical simulation of hypersonic turbulent boundary layers. Part IV: Effect of high enthalpy," *J. Fluid Mech.*, vol. 684, pp. 25–59, 2011b.

[13] L. Duan and M. Martin, "Assessment of turbulence and chemistry interaction in hypersonic turbulent boundary layers," *AIAA Journal*, vol. 49, no. 1, 2011.

[14] Y. Zhang, Y. Jia, and S. S. Wang, "A conservative multi-block algorithm for two-dimensional numerical model," *Journal of Physics and Mathematical Sciences*, vol. 1, no. 1, 2007.

[15] M. P. Katz, A. Almgren, M. B. Sazo, K. Eiden, K. Gott, A. Harpole, J. M. Sexton, D. E. Willcox, W. Zhang, and M. Zingale, "Preparing nuclear astrophysics for exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[16] P. Woodward and P. Colella, "The numerical simulation of two-dimensional fluid flow with strong shocks," *J. of Computational Physics*, vol. 54, no. 1, pp. 115–173, 1984.

[17] C. Yang, T. Kurth, and S. Williams, "Hierarchical roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 perlmutter system," *Concurrency and Comp.: Prac. and Exp.*, vol. 32, no. 20, p. e5547, 2020.

[18] A. Myers, A. Almgren, L. Amorim, J. Bell, L. Fedeli, L. Ge, K. Gott, D. P. Grote, M. Hogan, A. Huebl *et al.*, "Porting WarpX to GPU-accelerated platforms," *Parallel Computing*, vol. 108, p. 102833, 2021.

[19] C. Atkins and R. Deiterding, "Towards a Strand-Cartesian solver for modelling hypersonic flows in thermochemical non-equilibrium," in *23rd AIAA International Space Planes and Hypersonic Systems and Tech.s Conf.*, 2020.

[20] O. M. Browne, A. P. Haas, H. F. Fasel, and C. Brehm, "A nonlinear compressible flow disturbance formulation for adaptive mesh refinement wavepacket tracking in hypersonic boundary-layer flows," *Computers & Fluids*, p. 105395, 2022.

[21] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured AMR calculations using the SAMRAI framework," in *Proc. of the 2001 ACM/IEEE conf. on Supercomputing*, 2001, pp. 6–6.

[22] P. Colella, D. T. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. Van Straalen, "Chombo software package for AMR applications design document," *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)*, 2009.

[23] A. Bhatele, T. Gamblin, K. E. Isaacs, B. T. N. Gunney, M. Schulz, P.-T. Bremer, and B. Hamann, "Novel views of performance data to analyze large-scale adaptive applications," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, Nov. 2012, lLNL-CONF-554552. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389038

[24] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "BoxLib with tiling: An adaptive mesh refinement software framework," *SIAM J. on Scientific Comp.*, vol. 38, no. 5, pp. S156–S172, 2016.

[25] P. Mullowney, R. Li, S. Thomas, S. Ananthan, A. Sharma, J. S. Rood, A. B. Williams, and M. A. Sprague, "Preparing an incompressible-flow fluid dynamics code for exascale-class wind energy simulations," in *Proc. of the Int'l Conf. for High Performance Comp., Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476185

[26] L. Bertagna, O. Guba, M. A. Taylor, J. G. Foucar, J. Larkin, A. M. Bradley, S. Rajamanickam, and A. G. Salinger, "A performance-portable nonhydrostatic atmospheric dycore for the energy exascale earth system model running at cloud-resolving resolutions," in *Proc. of the Int'l Conf. for High Performance Comp., Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[27] M. Kronbichler, N. Fehn, P. Munch, M. Bergbauer, K.-R. Wichmann, C. Geitner, M. Allalen, M. Schulz, and W. A. Wall, "A next-generation discontinuous Galerkin fluid dynamics solver with application to high-resolution lung airflow simulations," in *Proc. of the Int'l Conf. for High Performance Comp., Networking, Storage and Analysis*, 2021, pp. 1–15.