Communications on Applied Mathematics and Computation
https://doi.org/10.1007/s42967-022-00235-9

ORIGINAL PAPER

Techniques, Tricks, and Algorithms for Efficient GPU-Based Processing of Higher Order Hyperbolic PDEs

Sethupathy Subramanian[1] · Dinshaw S. Balsara[1,2] · Deepak Bhoriya[1,3] · Harish Kumar[3]

Received: 27 May 2022 / Revised: 16 November 2022 / Accepted: 22 November 2022

Abstract
GPU computing is expected to play an integral part in all modern Exascale supercomputers. It is also expected that higher order Godunov schemes will make up about a significant fraction of the application mix on such supercomputers. It is, therefore, very important to prepare the community of users of higher order schemes for hyperbolic PDEs for this emerging opportunity.
Not every algorithm that is used in the space-time update of the solution of hyperbolic PDEs will take well to GPUs. However, we identify a small core of algorithms that take exceptionally well to GPU computing. Based on an analysis of available options, we have been able to identify weighted essentially non-oscillatory (WENO) algorithms for spatial reconstruction along with arbitrary derivative (ADER) algorithms for time extension followed by a corrector step as the winning three-part algorithmic combination. Even when a winning subset of algorithms has been identified, it is not clear that they will port seamlessly to GPUs. The low data throughput between CPU and GPU, as well as the very small cache sizes on modern GPUs, implies that we have to think through all aspects of the task of porting an application to GPUs. For that reason, this paper identifies the techniques and tricks needed for making a successful port of this very useful class of higher order algorithms to GPUs.
Application codes face a further challenge—the GPU results need to be practically indistinguishable from the CPU results—in order for the legacy knowledge bases embedded in these applications codes to be preserved during the port of GPUs. This requirement often makes a complete code rewrite impossible. For that reason, it is safest to use an approach based on OpenACC directives, so that most of the code remains intact (as long as it was originally well-written). This paper is intended to be a one-stop shop for anyone seeking to make an OpenACC-based port of a higher order Godunov scheme to GPUs.
We focus on three broad and high-impact areas where higher order Godunov schemes are used. The first area is computational fluid dynamics (CFD). The second is computational magnetohydrodynamics (MHD) which has an involution constraint that has to be mimetically preserved. The third is computational electrodynamics (CED) which has involution constraints and also extremely stiff source terms. Together, these three diverse uses of higher order Godunov methodology, cover many of the most important applications areas. In all three cases, we show that the optimal use of algorithms, techniques, and tricks, along with the use of OpenACC, yields superlative speedups on GPUs. As a bonus, we find a most remarkable and desirable result: some higher order schemes, with their larger operations count per zone,

Extended author information available on the last page of the article

show better speedup than lower order schemes on GPUs. In other words, the GPU is an optimal stratagem for overcoming the higher computational complexities of higher order schemes. Several avenues for future improvement have also been identified. A scalability study is presented for a real-world application using GPUs and comparable numbers of high-end multicore CPUs. It is found that GPUs offer a substantial performance benefit over comparable number of CPUs, especially when all the methods designed in this paper are used.

Keywords PDEs · Numerical schemes · Mimetic · High performance computing

Mathematics Subject Classification 65M08 · 65M22 · 76M12 · 35Q35 · 35Q61 · 35Q68 · 35Q85 · 76W05 · 68Q85

1 Introduction

The advent of GPUs with floating point support and error correction in hardware, as well as the advent of open standards for programming GPUs with OpenACC directives, could produce a sea change in Exascale computing. Almost every major high-performance computing (HPC) unit is on track to install an Exascale supercomputer in the next few years. Such supercomputers will consist of many nodes. Each node will have a multicore CPU feeding data to several on-node GPUs. It is expected that applications codes that use higher order Godunov-based algorithms will make up a significant fraction of the use cases for such Exascale supercomputers. As a result, it is very important, and very timely, to make a deep analysis of different algorithms that are available under the rubric of higher order Godunov (HOG) schemes. Some of these algorithms take well to GPU computing, but others can be seen from the very onset to be weaker options. Even with a favorable algorithmic mix, it is realized that GPUs do have their own inherent weaknesses and limitations. Even for the winning set of algorithms, it is very important to identify those techniques and tricks that allow the full potential of the GPU to be realized while skirting around the weaknesses of the GPU. The goals of this paper are, therefore, fourfold as follows.

i) Identify the winning combination of algorithms that result in a HOG scheme that will port successfully to GPUs.
ii) Identify the available set of techniques and tricks that allow these HOG algorithms to be (painlessly) ported over for GPU computing. This is done within the context of OpenACC so that the interested reader should be able to pick up the minimal set of directives needed for porting a HOG code to a GPU just by reading this one article.
iii) Show that this salubrious combination of optimal algorithm sets and the right set of techniques and tricks yields very attractive performance gains on modern GPUs.
iv) Show that these advantages extend to different hyperbolic systems with different collocations and constraints and treatment of stiff source terms. This gives diverse communities of scientists and engineers confidence that the innovations reported here are broad based and applicable to their own community.

To give the reader a foretaste of what is to come, we will show that as one goes to HOG schemes with increasing order of accuracy, the speedup in going from CPU to GPU actually improves (with some caveats). In other words (and under favorable circumstances), the GPU

is an ideal vehicle for offsetting the higher cost of an algorithm with greater computational complexity.

In this paper, we will focus on three different codes from three different application areas, all of which have been ported to GPUs. These codes represent the different styles in which HOG schemes are used in many prominent applications areas. The three focus areas, and our justification for choosing them, are the following.

i) We will focus on a code drawn from computational fluid dynamics (CFD). Such codes are workhorses in numerous engineering applications and also in aerospace and geophysical modeling arenas. CFD codes solve the compressible Euler equations and rely on a conventional HOG structure with zone-centered fluid variables that are updated with face-centered fluxes. CFD codes exemplify a traditional application of classical HOG schemes.
ii) The second code that we focus on is a code drawn from magnetohydrodynamics (MHD). Such codes contribute to applications in astrophysics, space physics, plasma physics, fusion and space re-entry vehicles. They rely on a solution of the compressible MHD equations, which have extremely non-linear and non-convex terms that can give rise to compound waves. The MHD equations have an inbuilt divergence constraint; i.e., the divergence of the magnetic field has to remain zero. This requires a face-centered collocation of the magnetic fields which are updated with edge-centered electric fields. As a result, an MHD code retains fluid variables that are collocated in the same way as a CFD code along with a divergence constraint-free evolution of the face-centered magnetic fields. Therefore, MHD codes exemplify a class of problems with extremely strong non-linearities and mimetic constraint preservation.
iii) The third code that we focus on is a code drawn from computational electrodynamics (CED). Such codes are very important in electrical and electronic engineering. Example scientific and engineering applications include: understanding the effects (and mitigation thereof) of solar storms upon power grids; further reductions of the radar cross section of military platforms; improvements in wireless communications and sensing by incorporating millimeter wave and terahertz technologies; sophisticated first-principles analysis and design of nanoscale plasmonics-based technologies for ultrafast optical computers and ultra-efficient photovoltaics; and even the development of novel optical microscopy techniques to reliably detect deadly human cancers at the earliest possible stage. CED codes evolve a combination of Faraday's law and the extended Ampere's law using modern upwind methods and constraint-preserving reconstruction. The magnetic induction and the electric displacement are both facially collocated and are updated by edge-collocated electric and magnetic fields. Besides the preservation of divergence constraints for both the facially collocated vector fields, such CED codes have to handle extremely stiff source terms because the conductivity of some materials can take on exceptionally large values. Therefore, CED codes exemplify a class of problems with mimetic constraint preservation and extremely stiff source terms.

All three codes in our application mix use ideas drawn from finite volume-type weighted essentially non-oscillatory (WENO) reconstruction that are adapted to the different collocations that are used. For each case, we will demonstrate the functioning of second-, third-, and fourth-order schemes so that we can demonstrate sustained GPU speedup at several orders of accuracy. In the next three paragraphs, we discuss each of these codes with a view to identifying the algorithmic sub-set that is best suited for GPU computing. When reading the next three paragraphs, the reader should realize that selecting GPU-friendly algorithms is like threading

the needle! Out of the many algorithmic possibilities for designing a CFD, MHD, or CED code, only a few satisfy all the requirements that we need for obtaining superlative performance on a GPU.

1.1 Identifying Optimal Algorithmic Combinations for GPU-Based Processing of HOG Schemes

CFD was indeed the first application area for which HOG schemes were developed (Godunov [26], van Leer [41], Woodward and Colella [42], Harten et al. [27], Shu and Osher [34, 35]). Early methods for CFD resorted to second-order reconstruction of zone-centered flow variables. In time, several authors (Jiang and Shu [29], Balsara and Shu [13], Balsara et al. [12], Balsara et al. [9]) provided very efficient formulations for implementing WENO reconstruction at all orders on structured meshes. When these reconstruction strategies are implemented directly for the conserved variables, the code can be written to absolutely minimize cache usage, which makes modern WENO reconstruction strategies one of the favored algorithmic elements for GPU computation. Progress was also made in the design of Riemann solvers (Rusanov [31], van Leer [41], Colella [19], Roe [30], Harten et al. [28], Einfeldt et al. [24], Toro et al. [39], Dumbser and Balsara [21]), to the point where modern Riemann solvers have a very low operation count and use cache memory very sparingly. Such Riemann solvers are to be favored in GPU environments because each core of a GPU has a very small cache. In particular, the Rusanov Riemann solver (Rusanov [31]), the HLL Riemann solver (Harten et al. [28]), the HLLC Riemann solver (Toro et al. [39]), and the HLLI Riemann solver (Dumbser and Balsara [21]) have very favorable cache usage.

It was realized early on that just the zone-centered reconstruction, coupled with application of Riemann solvers at the zone faces, would yield a very effective single stage of a multi-stage, high order in time, Runge-Kutta time stepping scheme (Shu and Osher [34], Shu [33]). But it is important to realize that in Exascale computation, patches of the mesh will be assigned to different nodes. Each patch will have to transfer boundary data to other patches that reside on other nodes. Consequently, Runge-Kutta schemes would require back and forth data transfers at each and every stage from host to device and vice versa, which is why they would be somewhat disfavored on GPUs. (Due to their immense popularity, we do nevertheless include Runge-Kutta schemes in the speed comparisons that we present in the results section.) Arbitrary derivatives (ADERs) in space and time schemes for higher order time stepping were designed based on Cauchy-Kovalevskaya methods (Titarev and Toro [37, 38], Toro and Titarev [40]) but those are again very memory intensive at higher orders, resulting in them being disfavored for GPU usage. In addition, for general hyperbolic systems, the Cauchy-Kovalevskaya procedure is unusually difficult to formulate for larger hyperbolic systems, with the result that it is not a general purpose method that will extend to all manner of hyperbolic PDEs. More recent incarnations of ADER schemes (Dumbser et al. [22], Balsara et al. [12]) are more suited to low cache usage and also generalize well to several different PDEs. For that reason, we will favor ADER time stepping schemes. To summarize the results of this paragraph, we arrive at a strategy for CFD that first relies on finite volume WENO reconstruction; followed by an ADER predictor step, and completed by a corrector step that invokes certain suitable Riemann solvers. Such a strategy, which is based on three conceptual stages—WENO reconstruction, ADER predictor, and Riemann solver-based corrector—was found to be an optimal approach for GPU computing. (ADER-WENO methods are extensively discussed in a review by Balsara [4].) In this approach, there is only one data transfer from host to node at the start of a timestep followed by a final data transfer at the end of a timestep from node to host.

Even this data transfer will be minimized, as we will see in Sect. 2.2. Also please note that due to the popularity and proliferation of Runge-Kutta-based time stepping, this option will also be explored in this paper.

In addition to the fluid variables, the MHD equations include Faraday's law for the evolution of the divergence-free magnetic field. Mimetic HOG schemes that preserve the divergence constraint have been designed (Ryu et al. [32], Dai and Woodward [20], Balsara and Spicer [14]). Such schemes collocate the normal components of the magnetic field vector at the faces of the mesh, and they are updated using the components of the electric field vector that is collocated at the edges of the mesh. Progress was made on the second-order reconstruction of the magnetic field in Balsara [2, 7] and the same was extended to higher orders using WENO-like methods in Balsara [3]. These WENO-like methods retain the advantage of minimizing cache usage, making them optimal for GPUs. ADER methods were first applied to MHD in Balsara et al. [10, 12] and these ADER methods also retain the same advantages as before. Evaluation of the edge-centered electric fields requires multidimensional upwinding at the edges and that was achieved by HLL-type multidimensional Riemann solvers (Balsara [5]) along with their HLLC variants (Balsara [1]) and their HLLI-type extensions (Balsara [6], Balsara and Nkonga [11]). These multidimensional HLL-style Riemann solvers retain the same advantages of cache friendliness as their one-dimensional versions, making them optimal for use on GPUs. Therefore, we see that all the algorithmic pieces that were identified for CFD find their extensions for MHD, making it possible to design optimal algorithmic combinations for use in GPU-enabled MHD codes.

CED shares some of the mimetic features with MHD in that one is now evolving two curl-type equations; one for Faraday's law and the other for the extended Ampere law. Face-centered collocations are, therefore, used for the normal component of the magnetic induction and electric displacement. These two vector fields are then updated using edge-centered components of the electric field and the magnetic field. WENO-like reconstruction strategies for the face-centered primal variables were documented in Balsara et al. [15, 16]. Multidimensional Riemann solvers of the HLL type for CED were also documented in Balsara et al. [15]. The CED equations incorporate stiff source terms by way of conductivities that can sometimes be millions of times larger than the contribution from the wave propagation terms. As a result, an optimal ADER formulation was documented in Balsara et al. [15, 16]. The ADER formulation is special because it allows the stiff source terms to be treated implicitly while treating the rest of the hyperbolic system explicitly. The implicit treatment requires the inversion of modest-sized matrices locally within each zone. Furthermore, the unique ADER formulation mentioned above decomposes the implicit treatment into a set of smaller block matrices whose inversion can be carried out independently. As a result, the matrix inversion is very easy, very tractable, though not very cache friendly. Consequently, we see that all the algorithmic pieces that were identified for CFD find their extensions for CED, making it possible to design optimal algorithmic combinations for use in GPU-enabled CED codes.

The above three paragraphs have shown how one (figuratively) can thread the algorithmic needle, arriving at end-to-end algorithmic ingredients for HOG schemes that are ideally suited for GPU processing. In Sect. 2, we describe how these are assembled. Section 3 shows results from GPU speedups for CFD, MHD, and CED codes on A100 GPUs. Section 4 offers some conclusions. Since the examples in Sect. 2 used Fortran-based pseudo code to exemplify the structure of GPU-friendly HOG schemes, in the Appendix we provide their C++equivalents. This gives practitioners a one-stop shop from which they can pick up the essentials of GPU programming for HOG schemes. Section 4 shows that the accuracy of the code is virtually unaffected by whether it is run on a GPU or a CPU; thereby giving us confidence that GPU computation should give us results that are entirely comparable in quality to the ones that we

have been used to getting on CPUs. Section 5 takes us through the process of setting up a real-world simulation on a GPU-rich supercomputer and carrying out a scalability study on such a machine. The exercise is intended to give us insight into the limitations as well as the opportunities in GPU computing. Section 6 presents some conclusions.

2 GPU Implementation

2.1 GPU Architecture and Programming Environment and the Resulting Constraints on Efficient Implementation

The trend in Exascale computing is to compute with the lowest power consumption, because the cost of keeping an Exascale supercomputer running can indeed be quite prohibitive. A majority of the end-to-end cost of owning and maintaining a supercomputer over its life cycle goes toward the cost of electricity. GPUs are the preferred pathway for Exascale computing because they couple performance with a level of (low) power consumption that cannot be matched by modern CPUs. As a result, a modern Exascale supercomputer will consist of many nodes that are connected together with a fast interconnect. Each such node will have a multicore CPU that is coupled with multiple high-end GPUs. The V100 and A100 GPUs from Nvidia are good exemplars of such GPUs; though at the time of this writing, AMD has also announced a GPU product. The compute power offered by one of those GPUs easily exceeds the compute power offered by a multicore CPU. The possibility of having four or six such GPUs per node means that the majority of on-node computation will be done by GPUs. It is for this reason that it is beneficial to very briefly discuss the architecture of a modern GPU, especially as it pertains to higher order Godunov schemes. Applications codes that use such schemes are anticipated to constitute a significant fraction of the workload on Exascale supercomputers, making the problem of optimal GPU performance very useful and timely.

The node is often referred to as the host because it orchestrates the majority of the data structure management on an Exascale supercomputer. The GPUs are referred to as the devices. There is a bus that connects the host to the node, but that bus is by necessity quite slow. As a result, the bus can act as a severe bottleneck and maintaining optimal performance requires managing data. Modern GPUs that are used for HPC can be thought of as extensive arrays of rather slow and weak processors (known as cores) with very limited cache memory. As a result, the cache has also to be used very efficiently. The fact that each GPU core functions as a slow and weak processor is not to be scoffed at, because a modern GPU can have as many as 5 000 such cores. Taken together, all the cores of a GPU can offer performance that is superior to a modern multicore CPU. What has made modern GPUs very desirable for HPC is the fact that the GPU cores also offer double precision floating point performance as well as error correction in hardware with the result that it is possible to expect that a result obtained on a GPU can be as accurate as a result obtained on a CPU.

It should also be mentioned that the amount of memory available on present-day GPUs is often capped off at ~ 80 GB, with the result that GPUs have much smaller integrated memory than the total RAM memory on a CPU node. As a result, an Exascale computation will have to be chunked into small chunks which are sent to the GPU for processing. For a structured mesh application, which we address here, this means that the node may hold multiple smaller chunks (or patches) of data, each of which is sent to the GPU for a time update. Because each patch will have to provide ghost zone information to neighboring patches, it is best to keep those patches resident on the host and have the host manage the MPI-based communication.

Optimal one-sided communication strategies have been discussed in Garain et al. [25] and will not be discussed here. The focus of our study in this paper is to optimize on-node GPU usage for higher order Godunov schemes.

Modern higher order Godunov schemes have a very intricate algorithmic structure and codes that encapsulate such algorithms can be many thousands of lines long. Some such codes can represent decades worth of effort and they need to reproduce certain well-known and well-calibrated benchmark computations. A complete code rewrite, for example in CUDA, is often unfeasible mainly because it would destroy the code's ability to reproduce the well-known and well-calibrated benchmarks. As a result, it is beneficial to find the quickest pathway to port those codes to GPUs with minimal rewriting of code. The advent of reasonably mature compilers that support OpenACC gives us good grounds for thinking that this can be accomplished. OpenACC is not a language, but rather, a set of language extensions. Similar to OpenMP, it has a set of directives that can direct the computer to express the parallelism that is inherent in an algorithm. Textbooks that describe OpenACC are available (Chandrasekaran and Juckeland [17]). For the sake of completeness, it should also be mentioned that OpenMP is also expected to have GPU support and textbooks for OpenMP (Chapman et al. [18]). Indeed, a code that is well structured for OpenMP can be very quickly adapted to OpenACC. It is our impression that OpenACC currently gives more fine-grained control over GPU computations and that the supporting compiler technology is also more mature. For this reason, we focus attention on GPU computation of higher order schemes using OpenACC directives. This is the topic for the rest of this section. The reader who is familiar with OpenMP will see many parallels between OpenMP and OpenACC; an OpenMP code can be an excellent starting point for a port to OpenACC.

2.2 Overall Structure of a Higher Order Godunov Scheme Implementation and Its Adaptation to GPUs

All HOG schemes of the ADER-WENO type have a conceptually identical structure; even if the different collocation of variables requires them to be different in detail. All such schemes begin with a spatial reconstruction of the primal variables with higher order accuracy. This first step examines the neighboring primal variables to build a higher order representation of the primal variables within each zone. This higher order representation is provided by the WENO or WENO-like reconstruction step. Once we know the spatial variation of all the variables in a hyperbolic PDE, the time-dependent structure of the PDE can itself be exploited to give us an in-the-small evolution of these variables in the temporal direction. This temporal evolution has to be sufficiently accurate so as to match the accuracy of the spatial reconstruction. This second step is provided by the ADER predictor step. The third step consists of a Riemann solver-based corrector step. In other words, the space-time evolved (predicted) solution of the hyperbolic PDE within each zone is made to interact with the analogous solutions from neighboring zones. This interaction is mediated by Riemann solvers which mediate the resolution of any jumps in the solution in a fashion that respects the physics of the PDE. This corrector step yields the numerical fluxes that are used for a space and time accurate update of the primal variables of the PDE. Because all ADER-WENO type algorithms for CFD, MHD, and CED share this same three-part structure, we will illustrate our ideas by instantiating them within the context of a second-order accurate CFD scheme.

The general structure of the HOG scheme is shown in Fig. 1, along with the necessary OpenACC directives. (While we use Fortran in the main body of the text, the figures in Appendix A show the exact C++equivalents, so that both major programming paradigms

Fig. 1 A hydro code (in pseudocode format) at second order with OpenACC extensions to allow it to perform well on a GPU. Notice that most of the code is identical to a serial code where we have a variable "*U*" which holds the five fluid components and their space-time variation. The data at the start of a timestep are located in "*U*(:, :, :, 1:5, 1)". These data are used to obtain the spatial variation which is stored in "*U*(:, :, :, 1:5, 2:4)" using the limiter routine. The predictor routine builds the temporal dependence in "*U*(:, :, :, 1:5, 5)". The Make_Flux routines make the appropriate fluxes and these are used in Make_d*U*_d*t* to make the time rate of change which is used in update to make the time update. These are all standard components of a classical hydro code. The changes that we introduced are shown using color. The blue variable "*U*_skinny" is the only minimal amount of data that is moved from host to device at the beginning of the timestep and then back from device to host at the end of a timestep. The OpenACC pragmas for carrying out this data motion are shown in red. An OpenACC pragma is also needed to tell the device to create requisite data on the device. The only new subroutine that we had to introduce is shown in magenta, called "*U*_Skinny_to_*U*". It operates on the device and takes the "*U*_Skinny" variable and copies it into "U(:, :, :, :, 1)" at the beginning of each timestep

are covered here.) The conserved variable "*U*" is dimensioned as "*U*(−1:*nx*+2, −1:*ny*+2, −1:*nz*+2, 5, 5)", here the first three dimensions correspond to the zones along the "*x*, *y*, *z*" directions, respectively. The active zones are numbered from "1:*nx*, 1:*ny*, 1:*nz*" and there are two ghost cells on the either side, which are necessary for a second-order accurate HOG scheme. The next dimension "5" corresponds to the 5 fluid components viz., density, *x*-momentum, *y*-momentum, *z*-momentum, and energy. The last dimension "5" stores the modes of each variable. The first mode corresponds to the zone-centered value; these are the primal variables that we seek to reconstruct, predict, and evolve in time. The second, third, and fourth modes correspond to the piecewise linear variations in the *x*-, *y*-, and *z*-directions. These are obtained via a WENO limiter step which provides the spatial reconstruction. The fifth mode corresponds to the temporal dependence of the hyperbolic PDE and is provided by the ADER predictor step. Along with this, there is another variable, which is a skinny version of the "*U*" variable, called "*U*_skinny", holds the zone-centered primal variables of "*U*", and this gets updated for every timestep. The "*U*_skinny" variable holds the minimal amount of data that is moved from the CPU (host) to GPU (device) at the beginning of the timestep and then back from the GPU (device) to CPU (host) at the end of the timestep.

The face-centered fluxes are stored in the variables "Flux_*X*, Flux_*Y*, and Flux_*Z*", respectively. For simplicity, they are all dimensioned as "(0:*nx*, 0:*ny*, 0:*nz*)", among which the *x*-face information is stored in locations "(0:*nx*, 1:*ny*, 1:*nz*)", *y*-face information is stored in locations "(1:*nx*, 0:*ny*, 1:*nz*)", and the *z*-face information is stored in locations "(1:*nx*, 1:*ny*, 0:*nz*)". For a zone located at "*i*, *j*, *k*", its left and right *x*-faces are indexed as "*i*−*1*, *j*, *k*" and "*i*, *j*, *k*", respectively and the corresponding fluxes are stored in the "Flux_*X*" variable. In the same way, its *y*-faces on the either are indexed as "*i*, *j*−1, *k*" and "*i*, *j*, *k*" and the corresponding fluxes are stored in the "Flux_*Y*" variable. Similarly, its *z*-faces on the either are indexed as "*i*, *j*, *k*−1" and "*i*, *j*, *k*" and the corresponding fluxes are stored in the "Flux_*Z*" variable. These fluxes will be filled in via a Riemann solver-based corrector step.

The time update computed from these fluxes for each zone is stored in "d*U*_d*t* (*nx*, *ny*, *nz*, 5)"; its first three dimensions correspond to the "*x*, *y*, and *z*" directions and the last dimension "5" holds the zone-centered time update corresponding to the 5 fluid variables. We also want to create storage on the device for several useful scalars. These scalars include the current timestep and the next timestep "d*t*, d*t*_next"; the Courant Friedrichs Levy number, "cfl" and the zone sizes "d*x*, d*y*, d*z*". The directive "!$acc data create" allocates the memory for these variables on the device. This is used to create the memory on the GPU for the variables "*U*_skinny, *U*, d*U*_d*t*, Flux_*X*, Flux_*Y*, Flux_*Z*". This directive does not copy the contents from CPU to GPU; it only creates the memory space for these variables on the GPU.

```
PROGRAM PSEUDO_HYDRO
INTEGER, PARAMETER :: nx = 48, ny = 48, nz = 48
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
REAL U_skinny (-1:nx+2, -1:ny+2, -1:nz+2, 5), dU_dt (nx, ny, nz, 5)
REAL, DIMENSION (0:nx, 0:ny, 0:nz, 5) :: Flux_X, Flux_Y, Flux_Z
REAL dt, dt_next, cfl, dx, dy, dz

number_of_timesteps = 500
!$acc data create (U_skinny, U, dU_dt, Flux_X, Flux_Y, Flux_Z,  &
!$acc&       dt, dt_next, cfl, dx, dy, dz)

CALL Initialize(nx, ny, nz, U_skinny, dt, cfl, dx, dy, dz)   ! CPU subroutine

!$acc update device (cfl, dx, dy, dz)

DO i_timestep = 1, number_of_timesteps

!$acc update device (U_skinny, dt)

CALL U_Skinny_to_U (nx, ny, nz, U_skinny, U)
CALL Boundary_Conditions (nx, ny, nz, U)
CALL Limiter (nx, ny, nz, U)
CALL Predictor (nx, ny, nz, U)
CALL Make_Flux_X (nx, ny, nz, U, Flux_X)
CALL Make_Flux_Y (nx, ny, nz, U, Flux_Y)
CALL Make_Flux_Z (nx, ny, nz, U, Flux_Z)
CALL Make_dU_dt (nx, ny, nz, Flux_X, Flux_Y, Flux_Z, dU_dt,
                 dt, dx, dy, dz)
CALL UPDATE_U_Timestep (nx, ny, nz, U, U_skinny, dU_dt, dt_next,
                        cfl, dx, dy, dz)
!$acc update host (U_skinny, dt_next)
dt = dt_next
END DO
!$acc end data
END PROGRAM PSEUDO_HYDRO
```

Once we have explained the data declaration and the use of various data variables in Fig. 1, it is valuable to describe the functional algorithmic steps in Fig. 1. It is expected that the reader has a nicely modularized CFD code, with the result that we only point attention to the additional steps that are needed to ready it for GPUs using OpenACC directives. Our goal, of course, is to show how easy and transparent this upgrade is and to identify the minimal set of OpenACC constructs needed to make this upgrade. In other words, we wish to provide the "techniques and tricks" so that anyone with a nicely modular code can make the transition to GPU usage in the quickest and most painless way possible. The general calling sequence of the

subroutines involved in a HOG scheme can be seen inside the timestep loop, which starts from "DO *i*_timestep = 1, number_of_timesteps" in Fig. 1. In this figure, the colorized text shows the newly added codes which are necessary for the GPU implementation. The red color shows the OpenACC directives, the blue color corresponds to the new data variable "*U*_skinny", and the magenta shows the new subroutine(s) or code needed for the GPU implementation. The sequence of the operations inside the time loop is as follows.

i) Update the device with the zone-centered values stored in the "*U*_skinny" variable. This is done with the "!$acc update device" directive in Fig. 1. We also copy some very useful scalars from host to device within the same directive. This ensures that the absolute minimum amount of data is moved from host to device. We will show later, in the results section, that not having this minimalist data transfer strategy can result in a significant performance penalty.
ii) Now that the minimal data have been moved over to the GPU, transfer the zone-centered variables from "*U*_skinny" to the first mode of "*U*". This is done in "*U*_Skinny_To_*U*".
iii) Apply the necessary boundary conditions on "*U*" via a call to "Boundary_Conditions". This ensures that physical values have been provided in the ghost zones.
iv) Reconstruct the "*x*, *y*, *z*" modes, i.e., "*U*(:, :, :, :, 2:4)", using a limiter. This is accomplished using a call to "Limiter". A nicely modularized HOG code should already have such a subroutine/function so that we only need to show the minimal OpenACC upgrades needed in such a limiter subroutine. (Indeed, the same upgrades would apply to a boundary condition subroutine, which is why we do not repeat such information for a boundary condition subroutine.)
v) Using the spatial variation of the primal variables, predict the in-the-small temporal evolution via a call to "Predictor". This mode is returned by the predictor step as data that are stored in "*U*(:, :, :, :, 5)". Any modularly written ADER-WENO code should have such a subroutine/function so that we only need to show the minimal OpenACC upgrades needed in such a predictor subroutine.
vi) Now that the spatial and temporal modes are built, compute the face-centered HLL/HLLEM fluxes via calls to Riemann solvers applied at the *x*-, *y*-, and *z*-faces. This is done via calls to the "Make_Flux_*X*, Make_Flux_*Y*, Make_Flux_*Z*" subroutines. As before, any modularly written ADER-WENO code should have such a subroutine/function so that we only need to show the minimal OpenACC upgrades needed in such a subroutine for evaluating fluxes.
vii) Next, compute the time rate of change of the primal variables from the fluxes. Again, we only need to show the minimal OpenACC upgrades that are needed.
viii) Lastly, update the "*U* (:, :, :, :, 1)" variable with "d*U*_d*t*". On a CPU, this would have completed the timestep. For a GPU implementation, we need to transfer the zone-centered values back to the "*U*_skinny" variable which is to be used for transferring the minimum amount of data back from device to host. This is done via the "!$acc update host" directive in Fig. 1. Using the same directive, we also copy over the next prognosticated timestep, "d*t*_next", for this particular patch of data. In a multi-processor setting, this timestep will be minimized across different patches on different nodes using an MPI reduction call; but providing that level of MPI-based detail is not within the scope of this paper. Section 5 does,

however, broadly discuss a scalability study on a GPU-rich supercomputer where MPI was used for the communication across nodes.

In the following subsections, the small number of OpenACC changes that are needed for each of the subroutines are discussed.

2.2.1 Subroutine—*U*_Skinny_to_*U*(···)

The "*U*_skinny" array stores the minimum amount of zone-centered data, which is to be transferred from the host to device and vice versa for each time step. This transfer of data is essential in a parallel setting to provide the ghost zone information to the neighboring patches; however, this MPI-based parallelism is not within the scope of this paper. Once the "*U*_skinny" values are updated on the device using "!$acc update device(*U*_skinny)" at the beginning of the time step, the values are then copied to the first mode of the conserved variable "*U*" in the subroutine "*U*_skinny_to_*U*(···)", as shown in Fig. 2. The triply nested loop is for iterating over the zones located along the *x*-, *y*-, and *z*-directions using the loop index variables "*i*, *j*, *k*". The fourth index corresponds to the five fluid variables.

There are mainly two OpenACC directives used in parallelizing the loops. The first one declares the parallel region of the OpenACC. The beginning of the parallel region is marked with "!$acc parallel" and the end of the parallel region is marked with "!$acc end parallel". The "vector_length(64)" specifies the length to be used in the vector operations on the GPU. Based on our experience and testing with different GPUs, a vector length 64 is chosen. (Depending on the structure of their code, the reader should use a profiling tool to identify

```
SUBROUTINE U_Skinny_to_U (nx, ny, nz, U_skinny, U)
INTEGER nx, ny, nz
REAL U_skinny (-1:nx+2, -1:ny+2, -1:nz+2, 5)
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
INTEGER i, j, k

!$acc parallel vector_length(64) present (U_skinny, U)
!$acc loop gang vector collapse (3) independent          &
!$acc& private (i, j, k)
DO k = -1, nz+2
  DO j = -1, ny+2
    DO i = -1, nx+2
    U (i, j, k, :, 1) = U_skinny (i, j, k, :)
    END DO
   END DO
END DO
!$acc end parallel

END SUBROUTINE U_Skinny_to_U
```

Fig. 2 Structure of the *U*_Skinny_to_*U* subroutine. The array "*U*_skinny" contains the minimum data that should be carried from the host to the device. This subroutine just copies the contents of "*U*_skinny" to "*U*(:, :, :, :, 1)". To exploit the maximum available amount of parallelism, it is imperative to collapse the threefold nested loop

the optimal vector length for their particular code.) The "present" clause is used to inform the GPU that the variables that are included in the "present" clause are already declared or copied to the GPU. Generally, the variables in the present clause are big arrays which are shared between the loops or smaller variables which remain constant during the loop execution. In this subroutine, the big arrays "*U*_skinny, *U*" are placed inside the present clause.

The second OpenACC directive, that is used here, declares the "loop" and specifies the nature of the parallelization used in the ensuing loop nest. Here, to utilize the parallelization to the maximum extent all the three loops are collapsed and vectorized with the help of the compiler. This is specified by the directive "!$acc loop gang vector collapse(3) independent". The last clause "independent" explicitly specifies to the compiler that the triply nested loop iterations are data independent and can be executed in parallel. In addition to this, the small variables that are private to the loops can be explicitly specified here, using the "private" clause. In this fashion, all of the shared and the private variables in a parallelizing loop are explicitly specified to the compiler.

Lastly, the end of the parallel region (which is also the end of the triply nested loops) is marked with the "!$acc end parallel". In the next subsection, directives used in the Limiter are described.

2.2.2 Subroutine—Limiter(···)

Figure 3 shows a simple monotone-centered limiter for the *x*-direction. For this, an inline function "MC_limiter" is used as an example. It takes the slopes from either side and the compression factor and gives back the *limited x*-slope. Apart from the triply nested DO loop, there is an outer loop with a loop induction variable "*n*", for each component of the fluid dynamics. It can be noted that the value of "*n*" changes each time, the "!$acc parallel" region is invoked. Also, the value of the "compression_factor" changes depending upon the fluid variable to which the limiter is applied, i.e., "*n*". (For example, one might want to use a steeper profile, i.e., a higher compression factor, for the density variable so that the code can represent contact discontinuities with reduced dissipation.) The efficient way to do this is to move the variables "*n*, compression_factor" to the GPU and then update the device when there is a change in their values. The first task is achieved by "!$acc data copyin"; this creates the memory for these variables on the GPU and copies the values. Inside the outer loop given by "DO $n = 1, 5$", once the values of "*n*, compression_factor" are changed, they are then updated on the device using "!$acc update device(*n*, compression_factor)". Once the use of these variables is complete, they need to be deleted from the device using "!$acc end data"; this is done at the end of the subroutine. Just to avoid any confusion about Fig. 3, please look at the last index in "*U*". It is the second index that is updated using the first index. In other words, "*U*(*i*, *j*, *k*, *n*, 2)", which holds the *x*-slope, is updated using "*U*(*i*, *j*, *k*, *n*, 1)" and its neighbors, which indeed hold the primal variables.

The OpenACC directives for the inner three loops are similar to the previous description. There are some specific points here which are worth noticing. In addition to the shared array variable "*U*", the variables "*n*, compression_factor" are placed in the "present" clause. These two variables remain constant for the parallel region. The private clause has two more real variables "*a*, *b*" which are private to the loop, and they are used in computing the *x*-slope. Each iteration of the parallel loop nest shown in Fig. 3 will, therefore, have its own copy of the

```
SUBROUTINE Limiter (nx, ny, nz, U)
INTEGER nx, ny, nz
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
INTEGER i, j, k, n
REAL s1, s2, compression_factor, MC_Limiter
MC_Limiter(a, b, cfac) = AMIN1(0.5*ABS(a+b), cfac*ABS(a),
                cfac*ABS(b)) * (SIGN(0.5,a) + SIGN(0.5,b))
!$acc data copyin (n, compression_factor)

DO n = 1, 5

IF (n == 1) compression_factor=2.0
IF (n >  1) compression_factor=1.5
!$acc update device (n, compression_factor)

!$acc parallel vector_length(64) present (n, U, compression_factor)
!$acc loop gang vector collapse (3) independent                    &
!$acc&    private (i, j, k, s1, s2)
 DO k = 0, nz+1
   DO j = 0, ny+1
     DO i = 0, nx+1
     s1 = U (i+1, j, k, n, 1) - U (i, j, k, n, 1)
     s2 = U (i, j, k, n, 1) - U (i-1, j, k, n, 1)
     U (i, j, k, n, 2) = MC_Limiter(s1, s2, compression_factor)
     END DO
    END DO
 END DO
!$acc end parallel
END DO
! Do similarly for other modes in the y- and z-directions
!$acc end data
END SUBROUTINE Limiter
```

Fig. 3 Application of a simple monotone-centered limiter in the x-direction to all the fluid variables. The inline function "MC_Limiter" is made to accommodate a compression factor which can indeed be changed based on the flow field being limited. (For example, one might want to provide a larger compression factor for the limiting of a density variable than for the other variables. The limited x-slopes are stored in "U(:, :, :, n, 2)". The limiter can also be applied in the y- and z-directions and the limited y- and z-slopes are stored in "U(:, :, :, n, 3)" and "U(:, :, :, n, 4)"

temporary variables "a, b" making them suitable for the computation of the undivided difference in that figure. A similar privatization should be used for all variables that are used for the WENO reconstruction at all other orders. Similarly, structured loops can be written for the y-,

z-slopes of the "U" variable, completing the execution of second-order accurate limiter. In the next subsection, parallelization of the predictor subroutine is described.

2.2.3 Subroutine—Predictor(···)

The predictor subroutine builds the fifth mode that contains the in-the-small time rate of change in the solution vector for the PDE. The general structure of the predictor subroutine, along with OpenACC directives, is shown in Fig. 4. For the higher-order schemes, the predictor step can be quite long and complicated; therefore, it is commonly packaged into a subroutine, which would operate once on each zone during each timestep. (We refer to it as a zone-wise subroutine.) The zone-wise subroutine is then called inside the triply nested loop. Its inputs are the first four modes of the five fluid variables; in other words, the zone-wise input "U_zonewise (1:5, 1:4) = U (i, j, k,1:5, 1:4)". Using the zone-centered value and the spatial modes, the "Predictor_Zonewise" subroutine builds the fifth mode; i.e., the time rate of change for all five fluid variables. This is then stored into the fifth mode of "U", i.e., in "U (i, j, k,1:5, 5)".

The OpenACC directives around the loop are exactly similar to that of the previous subroutines. It can be noted that the variable "U_zonewise" is a small 5×5 array, and it is classified as the private variable. (Incidentally, the use of such very small arrays also makes the pointwise predictor very cache friendly toward the small caches on the GPU.) By default, the OpenACC classifies an array variable as a shared variable, which is not true here. It is imperative that the "U_zonewise" is explicitly identified as a private variable for the GPU. The subroutine call inside the triply nested loop needs to be specified to the GPU as a sequential call, without any parallelization. This is accomplished with the help of the "!$acc routine seq" directive. This directive is placed inside the "Predictor_Zonewise" subroutine, as shown in the bottom of Fig. 4. In addition to this, the subroutines may be placed in different files during the compilation; therefore, an additional mention of this sequential calling needs to be present inside the "Predictor" subroutine. This is accomplished by the OpenACC directive in the third line of Fig. 4, that is, "!$acc routine(Predictor_Zonewise) seq". It informs the compiler that the "Predictor_Zonewise" subroutine is a purely sequential subroutine. In the next subsection, the flux evaluation subroutine is discussed.

2.2.4 Subroutine—Make_Flux_*X*(···)

The previous predictor step builds the spatial and temporal modes. Using those modes, the face-centered HLL/HLLEM fluxes can be built by calling the Riemann solvers at the x-, y-, and z-faces. This is done via calls to the "Make_Flux_X, Make_Flux_Y, Make_Flux_Z" subroutines. As an example, the general structure of the "Make_Flux_X" subroutine is shown in Fig. 5. Inside the triply nested loop, the conserved variables from the left (U_L) and right (U_R) zones at the face are constructed and sent to the "Riemann_Solver_Ptwise" to compute the resolved flux at the face. The values of the resolved flux are then copied to the bigger array "Flux_X" from the small pointwise array "flux_x_ptwise" for all the five fluid dynamic variables. (As with the predictor step, this pointwise processing ensures that each call to "Riemann_Solver_Ptwise" is independent of any other call. It also ensures that "Riemann_Solver_Ptwise" only uses very small private arrays making it very cache friendly toward the small caches on the GPU.)

```
SUBROUTINE Predictor (nx, ny, nz, U)
INTEGER nx, ny, nz
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
INTEGER i, j, k
REAL U_zonewise(5, 5)      ! local variable for each zone

!$acc routine(Predictor_Zonewise) seq

!$acc parallel vector_length(64) present (U)
!$acc loop gang vector collapse (3) independent     &
!$acc& private (i, j, k, U_zonewise)
 DO k = 0, nz+1
   DO j = 0, ny+1
     DO i = 0, nx+1
     U_zonewise (1:5, 1:4) = U (i, j, k, 1:5, 1:4)
     CALL Predictor_Zonewise (U_zonewise)  ! Obtain the time-mode
     U (i, j, k, 1:5, 5) = U_zonewise (1:5, 5)
     END DO
    END DO
 END DO
!$acc end parallel

END SUBROUTINE Predictor


SUBROUTINE Predictor_Zonewise (U_zonewise)
!$acc routine seq
! Pointwise predictor step
END SUBROUTINE Predictor_Zonewise
```

Fig. 4 Predictor step for building the fifth mode, i.e., the mode that contains the time rate of change. The zone-centered value "U(:, :, :, 1:5,1)" and the space modes "U(:, :, :, 1:5, 2:4)" that are obtained from the limiter are copied to a zone-wise variable called "U_zonewise" for all the five components of the hydrodynamics. Using these and the expression of fluxes the "Predictor_Zonewise" would compute the fifth mode, which contains the temporal evolution. This is then copied back to the fifth mode of the variable "U(:, :, :, 1:5, 5)" for all the five components of the hydrodynamics. Here, in the "!$acc routine(⋯) seq", the OpenACC declaration of the subroutine can be noted. It declares to the GPU that the subroutine would be called in a sequential manner. A similar statement needs to be made inside the "Predictor_Zonewise" subroutine and this is shown in the bottom part of the figure

As mentioned before, by default, the OpenACC classifies all array variables as shared variables, which is not the case for "U_L", "U_R", and "flux_x_ptwise". Therefore, it is imperative that the "U_L", "U_R", and "flux_x_ptwise" are explicitly identified as private variables

for the GPU. The same can be seen above the triply nested loop where all of the abovementioned variables, as well as the indices "*i*, *j*, *k*", are declared as "private" to the compiler. The pointwise variables "*U*_L", "*U*_R" are sent as input to the Riemann solver and the "flux_*x*_ptwise" is the output from the Riemann solver. The "Riemann_Solver_Ptwise" subroutine is executed sequentially inside the triply nested loop. Hence, the OpenACC directive "!$acc routine seq" is placed inside the subroutine as shown in the bottom of Fig. 5. In addition to this, the same is conveyed to the "Make_Flux_*X*" subroutine in the third line, using the directive "!$acc routine(Riemann_Solver_Ptwise) seq". In the next subsection, the building of the time rate of change of the conserved variable "*U*" is discussed.

2.2.5 Subroutine—Make_d*U*_d*t*(···)

The next step would be to build the time rate of change for the conserved variable "*U*" from the facial fluxes. This is carried out in the "Make_d*U*_d*t*" subroutine, as shown in Fig. 6. In addition to the facial flux variables, "Flux_*X*, Flux_*Y*, and Flux_*Z*", the zone sizes "d*x*, d*y*, d*z*" and the timestep size "d*t*" are needed to compute the evolution of "*U*". The variables "Flux_*X*, Flux_*Y*, Flux_*Z*, d*U*_d*t*" are already present on the GPU memory. The additional variables, "d*t*, d*x*, d*y*, d*z*" are now needed on the GPU; however, realize that they were already copied from the host to the device, using the directive "!$acc update device (···, d*t*, cfl, d*x*, d*y*, d*z*)" in Fig. 1. Once moved to the device, all of these variables can be placed inside the "present" clause as shared variables for the GPU. In the next subsection, the subroutine which updates the "*U*" with the time rate of change "d*U*_d*t*" is discussed.

2.2.6 Subroutine—Make_Update(···)

In this last step, the "*U* (:, :, :, :, 1)" variable is updated with "d*U*_d*t*" and the next prognosticated timestep, "d*t*_next" is evaluated. The general structure of this subroutine is shown in Fig. 7. The triply nested loop is used as before and the five fluid dynamic variables are updated for each zone. Notice that "cfl, d*x*, d*y*, d*z*" (which are all needed for evaluating the next timestep via "Eval_Tstep_Ptwise") are already present on the device. Since "d*t*_next" is reinitialized in this subroutine, it is updated on the device. The "REDUCTION (MIN: d*t*_next)" pragma tells the compiler to treat "d*t*_next" as a reduction variable that has to be reduced over all the cores of the GPU. This completes our description of the timestep.

As mentioned before, there is a need for the transfer of information from the device to the host at each time step. This is necessary during a parallel execution across many nodes, where the ghost cell information must be communicated (using MPI-based messaging) from one patch to the other at each time step. Even though, this is not within the scope of this paper, for the sake of completeness (and as a demonstration of a very popular use case), the minimum amount of data is packed into "*U*_skinny" and moved from host to device via the "!$acc update host(*U*_skinny)" directive in Fig. 1. To prepare for this, the updated zone-centered values of "*U*" are copied to the "*U*_skinny" variable in this subroutine, inside the triply nested loop. The prognosticated value for the next timestep, which should eventually be minimized

```
SUBROUTINE Make_Flux_X (nx, ny, nz, U, Flux_X)
INTEGER nx, ny, nz
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
REAL Flux_X (0:nx, 0:ny, 0:nz, 5)
INTEGER i, j, k
REAL flux_x_ptwise(5), U_L(5), U_R(5)

!$acc routine (Riemann_Solver_Ptwise) seq

!$acc parallel vector_length(64) present (U, Flux_X)
!$acc loop gang vector collapse (3) independent        &
!$acc& private (i, j, k, U_L, U_R, flux_x_ptwise)
 DO k = 1, nz
   DO j = 1, ny
     DO i = 0, nx
     U_L (:) = U (i, j, k, :, 1) + 0.5 * U (i, j, k, :, 2)
                                 + 0.5 * U (i, j, k, :, 5)
     U_R (:) = U (i+1, j, k, :, 1) - 0.5 * U (i+1, j, k, :, 2)
                                   + 0.5 * U (i+1, j, k, :, 5)
     CALL Riemann_Solver_Ptwise (U_L, U_R, flux_x_ptwise)
     Flux_X (i, j, k, :) = flux_x_ptwise (:)
     END DO
    END DO
 END DO
!$acc end parallel
END SUBROUTINE Make_Flux_X

SUBROUTINE Riemann_Solver_Ptwise (U_L, U_R, flux_x_ptwise)
!$acc routine seq
! Build the HLL/HLLEM flux at the x-face
END SUBROUTINE Riemann_Solver_Ptwise
```

Fig. 5 Pseudocode for the subroutine to build the flux along the x-faces. The "U_L" variable stores the value of "U" obtained from the zone which is at the left to the face and the "U_R" variable stores the value of "U" obtained from the zone which is at the right to the face. Thus, the facial values of the conserved variable "U" are stored in "U_L, U_R". This is then fed in to the Riemann solver subroutine to build the flux for the corresponding face. The Riemann solver subroutine is called in a serial fashion. The same has been indicated to the GPU in the third line of the figure. It declares to the GPU that the subroutine would be called in a sequential manner. A similar statement needs to be made inside the "Riemann_Solver_Ptwise" subroutine and this is shown in the bottom part of the figure. The pointwise fluxes obtained from the Riemann solver are stored in "Flux_X (0:nx, 1:ny, 1:nz, :)". The subroutines for building the "Flux_Y (1:nx, 0:ny, 1:nz, :), Flux_Z (1:nx, 1:ny, 0:nz,:)" can be written in the same way, as that of the subroutine shown here. For the y- and z-fluxes, instead of the second mode, the third and fourth modes of "U" will be used for building the "y" and "z" face fluxes, respectively

```
SUBROUTINE Make_dU_dt (nx, ny, nz, Flux_X, Flux_Y, Flux_Z,
                       dU_dt, dt, dx, dy, dz)
INTEGER nx, ny, nz
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
REAL, DIMENSION (0:nx, 0:ny, 0:nz, 5) :: Flux_X, Flux_Y, Flux_Z
REAL dU_dt (nx, ny, nz, 5)
REAL dt, dx, dy, dz
INTEGER i, j, k

!$acc parallel vector_length(64) present (               &
!$acc&    dt, dx, dy, dz, Flux_X, Flux_Y, Flux_Z, dU_dt)
!$acc loop gang vector collapse (3) independent          &
!$acc&    private (i, j, k)
 DO k = 1, nz
   DO j = 1, ny
     DO i = 1, nx
     dU_dt (i, j, k, :) =
           -  dt * (Flux_X (i, j, k, :) - Flux_X (i-1, j, k, :)) / dx
           -  dt * (Flux_Y (i, j, k, :) - Flux_Y (i, j-1, k, :)) / dy
           -  dt * (Flux_Z (i, j, k, :) - Flux_Z (i, j, k-1, :)) / dz
     END DO
    END DO
 END DO
!$acc end parallel
END SUBROUTINE Make_dU_dt
```

Fig. 6 The subroutine to build the dU_dt update from the fluxes obtained from the "Make_Flux_X, Make_Flux_Y and Make_Flux_Z" subroutines. The "dU_dt (1:nx, 1:ny, 1:nz,:)" variable is computed for all the active zones of the simulation. The common variables are present on the GPU and they are indicated as "present(⋯)"; the local variables are mentioned in the "private(⋯)" part

across nodes in a parallel implementation, is also ready to be sent over from the device to the host.

3 Results

In this section, we present results associated with GPU speedup for the CFD, MHD, and CED codes. All three codes are based on higher order Godunov scheme philosophy. In each case, we compare performance on an NVIDIA A100 PCIe GPU with 80 GB of memory relative to the performance of a single core of a 3.0 GHz Xeon Gold 6248R CPU. We used the NVIDIA NVHPC (version 22.2) Fortran compiler for all our runs with an optimization level of three.

```
SUBROUTINE UPDATE_U_Timestep (nx, ny, nz, U, U_skinny, dU_dt,
                              dt_next, cfl, dx, dy, dz)
INTEGER nx, ny, nz
REAL U (-1:nx+2, -1:ny+2, -1:nz+2, 5, 5)
REAL U_skinny (-1:nx+2, -1:ny+2, -1:nz+2, 5), dU_dt (nx, ny, nz, 5)
REAL dt_next, cfl, dx, dy, dz
INTEGER i, j, k
REAL U_ptwise(5), dt1

!$acc routine(Eval_Tstep_Ptwise) seq

dt_next = 1.0e32
!$acc update device(dt_next)

!$acc parallel vector_length(64) present (U,            &
!$acc&   U_skinny, dU_dt, cfl, dx, dy, dz )
!$acc loop gang vector collapse (3) independent         &
!$acc&    private (i, j, k, U_ptwise, dt1)
!$acc&    REDUCTION (MIN: dt_next)
DO k = 1, nz
  DO j = 1, ny
    DO i = 1, nx
    U (i, j, k, :, 1)=U (i, j, k, :, 1)+ dU_dt (i, j, k, :)
    U_skinny (i, j, k, :) = U (i, j, k, :, 1)
    U_ptwise (:) = U (i, j, k, :, 1)
    CALL Eval_Tstep_Ptwise (cfl, dx, dy, dz, U_ptwise, dt1)
    dt_next = MIN(dt_next, dt1)
    END DO
  END DO
END DO
!$acc end parallel
END SUBROUTINE UPDATE_U_Timestep
```

Fig. 7 The subroutine to update the "*U*" from the "d*U*_d*t*". In addition to this, here the "*U*_skinny" is updated from the new zone-centered values of "*U*". The minimum amount of data stored in the "*U*_skinny" will be moved back to the CPU, demonstrating the importance of minimal data movement in a CPU + GPU computer. In a parallel setting, this data can be used to provide ghost zone information to neighboring patches; but that is not within the scope of this paper. Also in this function, the next timestep "d*t*_next" is estimated, using the CFL number, zone size, and *U*. The smallest among all the zones is found using the "MIN()" function. To correctly parallelize this reduction operation, the OpenACC command, "REDUCTION(MIN: d*t*_next)" is used

This compiler works on CPUs and it also has full-featured OpenACC which enables it to work on GPUs.

Table 1 A100 GPU speedup for the CFD code when the "*U*_skinny" trick was not used

Zones	Serial time/s	Zones/ Serial time/s	GPU time/s	Zones / GPU time/s	Speed up
O2					
48^3	1.43E−01	7.75E+05	8.43E−03	1.31E+07	16.9
96^3	1.14E+00	7.75E+05	3.98E−02	2.22E+07	28.7
144^3	3.67E+00	8.14E+05	1.10E−01	2.72E+07	33.5
192^3	8.73E+00	8.11E+05	2.62E−01	2.70E+07	33.3
O3					
48^3	3.37E−01	3.29E+05	2.56E−02	4.32E+06	13.1
96^3	2.41E+00	3.67E+05	1.08E−01	8.18E+06	22.3
144^3	7.85E+00	3.81E+05	3.31E−01	9.03E+06	23.7
192^3	1.86E+01	3.81E+05	7.99E−01	8.86E+06	23.2
O4					
48^3	1.13E+00	9.75E+04	5.75E−02	1.92E+06	19.7
96^3	7.04E+00	1.26E+05	3.43E−01	2.58E+06	20.5
144^3	2.23E+01	1.34E+05	1.14E+00	2.61E+06	19.5
192^3	5.14E+01	1.38E+05	2.97E+00	2.39E+06	17.3

3.1 CFD Results

Tables 1 and 2 show the GPU performance that is run on an A100 GPU. In each case, we compare the single core CPU performance with the GPU performance. (Section 5 will show cases where the speeds of several GPUs are compared to the speeds equivalent numbers of multicore CPUs for a real-world application.) Table 1 shows the results when the "*U*_skinny" trick is not used; Table 2 shows the results when the "*U*_skinny" trick is indeed used. Table 1 shows the traditional amount of speedup that is often reported by several practitioners for GPUs; indeed it is fair to say that it is encouraging but not very impressive. We see a dramatic improvement in the results shown in Table 2, and we see that the speedup is indeed quite compelling. This shows the importance of only making minimal data motion between the host and the device. From Table 2, we also see that the third-order scheme shows better speedup on comparable meshes than the second-order scheme. This is an advantage that we will see persisting even in MHD calculations. The CFD code and the MHD code use the similar type of WENO, ADER, and Riemann solver algorithms; therefore, their speedup trend lines are similar. Table 2 also demonstrates that with all the right algorithmic choices, along with deft utilization of OpenACC pragmas, it is indeed possible to unlock the potential of GPU computing for CFD applications.

It is important to point out another issue. Please look at the raw numbers for zones updated per second on the CPU in Table 2. We see that at the second order, the CFD code updates ~ 0.77 million zones per second on a CPU. At the third order, the CFD code updates ~ 0.377 million zones per second on a CPU. At the fourth order, the CFD code updates ~ 0.136 million zones per second on a CPU. These are very good speeds for a CFD code on a CPU, showing that the CPU code was already highly optimized for CPUs. On CPUs, we have made efforts to ensure that the code is optimally cache friendly. It is harder to show good GPU speedups when the code is already optimized for CPUs. The fact that Table 2, nevertheless, shows good speedup

Table 2 A100 GPU speedup for the CFD code when the "*U*_skinny" trick was used

Zones	Serial time/s	Zones/ Serial time/s	GPU time/s	Zones/ GPU time/s	Speed up
O2					
48^3	1.43E−01	7.73E+05	3.43E−03	3.22E+07	41.7
96^3	1.13E+00	7.86E+05	1.58E−02	5.60E+07	71.3
144^3	3.71E+00	8.04E+05	4.62E−02	6.46E+07	80.4
192^3	9.11E+00	7.77E+05	1.08E−01	6.55E+07	84.4
O3					
48^3	3.32E−01	3.33E+05	5.17E−03	2.14E+07	64.3
96^3	2.47E+00	3.58E+05	2.47E−02	3.58E+07	100.0
144^3	7.90E+00	3.78E+05	7.30E−02	4.09E+07	108.2
192^3	1.88E+01	3.77E+05	1.72E−01	4.12E+07	109.3
O4					
48^3	1.11E+00	9.98E+04	1.70E−02	6.49E+06	65.0
96^3	7.14E+00	1.24E+05	9.24E−02	9.58E+06	77.4
144^3	2.24E+01	1.33E+05	2.85E−01	1.05E+07	78.7
192^3	5.21E+01	1.36E+05	6.21E−01	1.14E+07	83.8

on a GPU means that we had to do everything right even on the GPU to extract that level of speedup.

Comparing the speedups at the second and third orders in Table 2, we see that the third-order scheme shows substantially better speedup on the GPU than on the CPU. This is because the data motion from CPU to GPU is the same for both schemes; however, the third-order scheme uses more floating point operations per zone than the second-order scheme. The fourth-order scheme has a comparable speedup to the second-order scheme, but the speedup is not so good when compared to the third-order scheme. This is explained in Table 3 where we focus exclusively on the fraction of time that the CFD code spends within the ADER predictor step. At the second and third orders, the fraction of time spent within the ADER predictor step is comparable on the CPU and on the GPU. Admittedly, the third-order ADER takes the place of a three-stage third-order Runge-Kutta scheme while the second-order ADER scheme takes the place of a two-stage second-order Runge-Kutta scheme. As a result, the fraction of time spent in the ADER step is larger for the third-orderscheme compared to the second-order scheme. But now please compare the fractional times spent in the ADER predictor step at the fourth order on CPU and GPU. We see that the GPU code spends a much larger fraction of its time in the ADER predictor step compared to the CPU code. The physical reason for that is that ADER predictor steps use increasingly larger amounts of memory with increasing order. The CPU has a large cache, so it works fine with higher order ADER. However, the GPU cores have a very small cache, so the fourth-order ADER scheme is not as adept at using the (smaller) cache on the GPU cores very efficiently. At the point of this writing, we do not know of any strategies to reduce the cache usage of higher order ADER predictor steps. Even so, this work serves to identify one of the bottlenecks in present-day GPU computing.

The curious reader might still want to know what would happen if we had used an equivalent Runge-Kutta temporal update strategy. This is explored in Table 4. (Please recall that a second-order Runge-Kutta scheme uses two sub-steps; a third-order Runge-Kutta scheme uses three sub-steps but a fourth-order Runge-Kutta scheme uses five sub-steps.) The fifth column

Table 3 Comparing the fractional time spent in the ADER subroutine at various orders on CPU and GPU for the CFD code

Zones	Serial time/s	Fraction of time in ADER	GPU time/s	Fraction of time in ADER
O2				
48^3	4.48E−02	0.313	3.49E−04	0.102
96^3	2.82E−01	0.250	1.99E−03	0.126
144^3	8.76E−01	0.236	6.13E−03	0.133
192^3	2.03E+00	0.222	1.40E−02	0.130
O3				
48^3	1.62E−01	0.488	1.76E−03	0.340
96^3	1.10E+00	0.447	1.07E−02	0.433
144^3	3.42E+00	0.433	3.30E−02	0.451
192^3	7.87E+00	0.419	7.52E−02	0.437
O4				
48^3	6.19E−01	0.559	1.29E−02	0.755
96^3	3.78E+00	0.528	7.32E−02	0.792
144^3	1.12E+01	0.501	2.20E−01	0.772
192^3	2.56E+01	0.491	4.91E−01	0.791

Table 4 A100 GPU speedup for the CFD code where we exclusively focus on the performance differences between the ADER-based scheme and the Runge-Kutta-based scheme

Zones	ADER-CPU Zones/s	RK-CPU Zones/s	RK-GPU Zones/s	Speed up ADER-CPU/ ADER-GPU	Relative Speed up RK-GPU/ ADER-CPU	Speed up RK-GPU/ RK-CPU
O2						
48^3	7.73E+05	5.56E+05	1.61E+07	41.7	20.88	29.04
96^3	7.86E+05	5.02E+05	3.08E+07	71.3	39.15	61.29
144^3	8.04E+05	5.26E+05	3.60E+07	80.4	44.72	68.43
192^3	7.77E+05	4.99E+05	3.66E+07	84.4	47.08	73.33
O3						
48^3	3.33E+05	1.92E+05	1.03E+07	64.3	30.89	53.58
96^3	3.58E+05	2.01E+05	1.86E+07	100.0	51.83	92.58
144^3	3.78E+05	2.04E+05	2.08E+07	108.2	54.95	101.73
192^3	3.77E+05	1.99E+05	2.05E+07	109.3	54.42	102.87
O4						
48^3	9.98E+04	3.98E+04	5.05E+06	65.0	50.59	126.97
96^3	1.24E+05	4.68E+04	8.43E+06	77.4	68.04	179.95
144^3	1.33E+05	4.72E+04	9.67E+06	78.7	72.67	204.95
192^3	1.36E+05	4.75E+04	9.59E+06	83.8	70.51	201.65

in Table 4 just repeats the last column from Table 2. The sixth column compares Runge-Kutta speed on a GPU to ADER speed on a CPU. The seventh column in Table 4 compares the Runge-Kutta speed on a GPU to the Runge-Kutta speed on a CPU. Comparing the fifth and sixth columns in Table 4, we see that the speedup of a Runge-Kutta-based code on the GPU

relative to the CPU speed of an ADER-based code is not very impressive. The last column of Table 4 compares the speedup of a Runge-Kutta-based code on the GPU relative to the same code on a CPU. Then indeed we see quite wonderful speedups in the last column of Table 4. There are two ways to interpret this column. First, if all that the user has is a Runge-Kutta code, then this column shows that the GPU speeds will be vastly better than CPU speeds, as long as the "*U*_skinny" trick is used. This is a strong positive and gives us a pathway to dramatically improve the speeds of Runge-Kutta-based codes. Second, we realize that the Runge-Kutta time stepping algorithm does not use an ADER step, which was found to be a bottleneck in Table 3. Therefore, the last column of Table 4 also shows us that with increasing order, and all the way up to the fourth order, a Runge-Kutta-based code will show improving speedup with increasing order. However, note that the intrinsic number of zones updated per second by a Runge-Kutta-based CFD code is consistently lower compared to an ADER-based CFD code. (Compare the fourth column in Table 4 to the fifth column in Table 2.) This trend is true for CPUs and GPUs, and shows us that a one-time investment in ADER algorithms can be worthwhile.

3.2 MHD Results

Tables 5 and 6 show the speedups associated with an MHD code that was run in two possible configurations. In the first configuration, we ran it on the GPU without the "*U*_skinny" trick from Sect. 2, and those results are shown in Table 5. In the second configuration, we used the "*U*_skinny" trick and ran it on the GPU.

Table 5 clearly shows that as we go to higher orders, the speedup suffers. This is because the "*U*" variable itself is sent from host to node at the beginning of the timestep and it is sent back from node to host at the end of the timestep. This turned out to be a very bad choice because of the performance degradation. We, nevertheless, show it here because it

Table 5 A100 GPU speedup for the MHD code when the "*U*_skinny" trick was not used

Zones	Serial time/s	Zones/ Serial time/s	GPU time/s	Zones / GPU time/s	Speed up
O2					
48^3	8.23E−01	1.34E+05	2.53E−02	4.37E+06	32.5
96^3	6.27E+00	1.41E+05	1.07E−01	8.24E+06	58.4
144^3	2.08E+01	1.44E+05	3.35E−01	8.92E+06	62.1
192^3	5.12E+01	1.38E+05	7.36E−01	9.62E+06	69.6
O3					
48^3	1.35E+00	8.19E+04	5.09E−02	2.17E+06	26.5
96^3	1.08E+01	8.16E+04	2.76E−01	3.21E+06	39.4
144^3	3.72E+01	8.04E+04	7.85E−01	3.81E+06	47.3
192^3	9.25E+01	7.66E+04	1.69E+00	4.19E+06	54.7
O4					
48^3	3.06E+00	3.62E+04	1.35E−01	8.21E+05	22.7
96^3	2.28E+01	3.87E+04	7.34E−01	1.21E+06	31.1
144^3	7.86E+01	3.80E+04	2.57E+00	1.16E+06	30.6
192^3	1.89E+02	3.75E+04	4.25E+00	1.66E+06	44.3

Table 6 A100 GPU speedup for the MHD code when the "*U*_skinny" trick was used

Zones	Serial time/s	Zones/ Serial time/s	GPU time/s	Zones/ GPU time/s	Speed up
O2					
48^3	8.01E−01	1.38E+05	1.66E−02	6.65E+06	48.2
96^3	6.18E+00	1.43E+05	6.71E−02	1.32E+07	92.2
144^3	2.09E+01	1.43E+05	2.13E−01	1.40E+07	98.1
192^3	5.26E+01	1.34E+05	4.66E−01	1.52E+07	112.9
O3					
48^3	1.33E+00	8.29E+04	2.33E−02	4.74E+06	57.2
96^3	1.05E+01	8.41E+04	1.01E−01	8.74E+06	103.9
144^3	3.81E+01	7.83E+04	3.18E−01	9.38E+06	119.8
192^3	9.43E+01	7.50E+04	7.59E−01	9.33E+06	124.3
O4					
48^3	3.09E+00	3.58E+04	5.16E−02	2.15E+06	60.0
96^3	2.28E+01	3.87E+04	2.62E−01	3.37E+06	87.1
144^3	7.80E+01	3.83E+04	8.29E−01	3.60E+06	94.1
192^3	1.89E+02	3.74E+04	1.87E+00	3.78E+06	100.9

will be very natural for several CPU-based codes to be written in this fashion, and unless the appropriate trick is used to make a small modification of the code, the GPU advantage will not be realized. Table 6 shows the results from the GPU after the "*U*_skinny" trick was implemented correctly. We see that even on rather small meshes of 96^3 or 144^3 zones, a very admirable speedup is achieved.

We also begin to see another very attractive feature in Table 6, which is that the higher order schemes show better speedup than the lower order schemes. In other words, a higher order MHD scheme naturally entails more floating point operations per zone than a lower order scheme. However, when comparing the third-order results to the second-order results in Table 6, we see that the number of zones updated per second on the GPU for both schemes is quite competitive. This is because the third-order MHD code is more floating point intensive, with the result that it utilizes the computational capabilities of the GPU cores much more efficiently. The similar trend is unfortunately not fully realized when we compare the fourth-order results to the third-order results in Table 6; and we will identify the reason in Table 7. The tables also show that the GPU speedups only become attractive when sufficient work is given to the GPU. For example, when meshes with 48^3 zones were given to the GPU, the performance improvement from the GPU was not quite so dramatic. It is only at 96^3 or 144^3 zones per patch that we see dramatic improvements in the GPU performance relative to the CPU performance in Table 6. This tells us that rather large patches of MHD mesh need to be processed on a GPU before the advantage of the GPU is realized.

Table 7 brings out one of the shortcomings of the GPU. Realize that the caches available on each GPU core are significantly smaller than the caches available on the CPU cores. Conceptually, all the MHD codes explored here can be viewed as a WENO reconstruction, followed by an ADER predictor step and then followed by a Riemann solver-based corrector step. We were able to write the WENO algorithm in a way that has the smallest possible memoryfootprint in the processor's cache. We were also able to write the HLL Riemann solver so that it has the smallest possible memory footprint in the processor's cache. These

Table 7 Comparing the fractional time spent in the ADER subroutine at various orders on CPU and GPU for the MHD code

Zones	Serial time/s	Fraction of time in ADER	GPU time/s	Fraction of time in ADER
O2				
48^3	6.24E−02	0.078	5.62E−04	0.034
96^3	3.93E−01	0.064	3.44E−03	0.051
144^3	1.22E+00	0.058	1.06E−02	0.050
192^3	2.87E+00	0.055	2.42E−02	0.052
O3				
48^3	2.29E−01	0.172	4.45E−03	0.191
96^3	1.50E+00	0.143	2.76E−02	0.272
144^3	4.81E+00	0.126	8.58E−02	0.270
192^3	1.10E+01	0.117	1.95E−01	0.257
O4				
48^3	8.11E−01	0.262	2.77E−02	0.537
96^3	4.84E+00	0.212	1.59E−01	0.607
144^3	1.47E+01	0.188	4.81E−01	0.580
192^3	3.37E+01	0.178	1.07E+00	0.573

two improvements caused significant speedups in the CPU and GPU performance of the code. The ADER step that we used for MHD is based on the ADER algorithm described in Balsara et al. [10, 12] and already uses the smallest possible arrays that are needed to implement the ADER algorithm within each zone. The ADER algorithm, therefore, utilizes the CPU's larger cache very efficiently and is very good at allowing us to make cache-resident calculations on any modern CPU. However, recall that the GPU cores have significantly smaller caches. As the order of the ADER scheme increases, the temporary arrays that it needs also utilize more memory. This memory utilization is small at all orders on a CPU; however, it is not small compared to the cache on a GPU. For that reason, Table 7 shows just the ADER performance at various orders on CPU and GPU. We see that the ADER performance at the second order remains excellent on the GPU, because the second-order ADER algorithm can fit inside the GPU's cache. At the third order, Table 7 shows a slight degradation in the ADER performance relative to second order. At the fourth order, Table 7 shows an even greater degradation in the ADER performance relative to third order. This is because, with increasing order of accuracy, the ADER algorithm utilizes the very tiny GPU cache in an increasingly inefficient way. While this is not very exciting for the application developer, it highlights a direction in which GPU architectures can be improved in order for GPUs to become more broadly useful.

Another way to demonstrate some of the points made in the above paragraph would be to compare the times for the whole code in Table 6 to the time taken for just the ADER update in Table 7. The fraction of time spent in the ADER predictor step is shown in Table 7. At the second order, and on larger meshes with 96^3 or 144^3 zones, we see that the CPU and GPU spend about the same fraction of time in the ADER predictor step. At the third order, and again on larger meshes with 96^3 or 144^3 zones, we see that the fraction of time spent in the ADER step increases on the GPU compared to the CPU. At the fourth order, and again on larger meshes with 96^3 or 144^3 zones, we now see that the fraction of time spent in the ADER step is again reasonably constant for the CPU. However, the

fraction of time spent in the ADER step on the GPU increases. It might be possible to hard code the ADER algorithm at each order for GPUs, but that would diminish its portability and malleability of usage. We refer to this as the "Predictor bottleneck" because it is currently not possible to arrive at general-purpose predictor algorithms at all orders where the implementation is small enough to be cache resident on the very small GPU caches of present-day GPUs.

As with Table 4 in the previous subsection, Table 8 in this section considers Runge-Kutta-based schemes. The conclusion from either of these tables is the same. The Runge-Kutta-based MHD code is an inferior performer on GPUs when compared to the ADER-based MHD code. However, the last column of Table 8 mirrors the last column of Table 4. It shows that as we go to higher orders, a Runge-Kutta-based code on the GPU will show impressive speedups relative to the Runge-Kutta-based code on the CPU. Furthermore, this speedup improves as we go to higher order as long as the "*U*_skinny" trick is used (Table 10).

3.3 CED Results

There is an essential, and structural, difference between CFD and MHD codes on the one hand and CED codes on the other hand. For CFD and MHD codes, the WENO-based reconstruction step, the ADER-based predictor step, and the Riemann solver-based corrector step require roughly comparable amounts of floating point operations. In a CED code, the WENO-based reconstruction step has to be entirely divergence preserving (Balsara and Shu [13] or Balsara et al. [16]) which makes it very inexpensive. Furthermore, the multi-dimensional Riemann solver used in the corrector step of a CED code is extremely inexpensive (see Eqns. (5.5) and (5.6) from Balsara et al. [15]). As a result, the reconstruction and the corrector steps in a CED code are unusually lightweight and require very few floating point operations. However, realize too that most CED applications do have unusually large conductivities, which results in very stiff source terms. ADER schemes that treat stiff source terms (Dumbser et al. [23], Balsara et al. [16]) have been designed, but they are considerably more memory and floating point intensive than ADER schemes for treating problems without stiff source terms. While diagonally implicit Runge-Kutta schemes have also been used for treating stiff source terms in CED (Balsara et al. [8]), they tend to be even more inefficient than ADER schemes when used in finite volume context for CED applications. (This is why we do not show any Runge-Kutta-based results for CED.) Thus, CED is a harsher algorithmic terrain for GPU computing compared to CFD and MHD.

Tables 9 and 10 show the performance of CED codes when the "*U*_skinny" trick is not used and when it is used. We see from Table 10 that the trick does have a rather positive effect on the overall speedup. However, the speedups are not as impressive as the ones that we have seen in prior sections for CFD and MHD. This is understandable considering that the ADER predictor step is so dominant for CED codes, as we will see in the next paragraph.

Table 11 focuses on the fractional time taken by the ADER predictor step for CED on CPUs and GPUs at various orders. It gives us greater clarity of perspective. At the second order, the ADER predictor step is still quite light weight and takes up comparable fractions of time on the CPU and the GPU. In other words, at the second order, it is possible to arrive at cache-friendly versions of the ADER predictor step on both the CPU and the GPU. However, at the third order, we see that the fraction of time taken in the ADER predictor step is considerably larger on the GPU than on the CPU. This tells us that at the

Table 8 A100 GPU speedup for the MHD code where we exclusively focus on the performance differences between the ADER-based scheme and the Runge-Kutta-based scheme

Zones	ADER-CPU zones/s	RK-CPU zones/s	RK-GPU zones/s	Speedup ADER-CPU/ ADER-GPU	Relative speedup RK-GPU/ ADER-CPU	Speedup RK-GPU/ RK-CPU
O2						
48^3	1.38E+05	7.53E+04	3.42E+06	48.2	24.77	45.42
96^3	1.43E+05	7.60E+04	7.06E+06	92.2	49.36	92.98
144^3	1.43E+05	7.71E+04	7.42E+06	98.1	51.96	96.25
192^3	1.34E+05	7.33E+04	8.12E+06	112.9	60.42	110.80
O3						
48^3	8.29E+04	3.28E+04	1.97E+06	57.2	23.70	60.00
96^3	8.41E+04	3.22E+04	3.93E+06	103.9	46.78	122.23
144^3	7.83E+04	3.03E+04	4.23E+06	119.8	54.06	139.59
192^3	7.50E+04	2.78E+04	4.39E+06	124.3	58.55	158.12
O4						
48^3	3.58E+04	8.96E+03	9.29E+05	60.0	25.97	103.69
96^3	3.87E+04	9.44E+03	1.70E+06	87.1	43.94	180.25
144^3	3.83E+04	9.11E+03	1.74E+06	94.1	45.36	190.63
192^3	3.74E+04	8.88E+03	1.77E+06	100.9	47.20	199.12

Table 9 A100 GPU speedup for the CED code when the "*U*_skinny" trick was not used

Zones	Serial time/s	Zones/Serial time/s	GPU time/s	Zones/GPU time/s	Speedup
O2					
48^3	4.09E−01	2.71E+05	1.20E−02	9.23E+06	34.1
96^3	2.69E+00	3.29E+05	7.63E−02	1.16E+07	35.2
144^3	9.94E+00	3.00E+05	2.37E−01	1.26E+07	41.9
192^3	2.17E+01	3.26E+05	5.21E−01	1.36E+07	41.8
O3					
48^3	1.19E+00	9.31E+04	3.41E−02	3.24E+06	34.8
96^3	8.19E+00	1.08E+05	2.14E−01	4.14E+06	38.3
144^3	2.64E+01	1.13E+05	6.73E−01	4.44E+06	39.2
192^3	6.41E+01	1.10E+05	1.46E+00	4.84E+06	43.8
O4					
48^3	5.74E+00	1.93E+04	1.24E−01	8.88E+05	46.1
96^3	3.72E+01	2.38E+04	7.15E−01	1.24E+06	52.0
144^3	1.12E+02	2.67E+04	2.13E+00	1.41E+06	52.7
180^3	2.08E+02	2.81E+04	3.98E+00	1.46E+06	52.2

Table 10 A100 GPU speedup for the CED code when the "*U*_skinny" trick was used

Zones	Serial time/s	Zones/ Serial time/s	GPU time/s	Zones/ GPU time/s	Speedup
O2					
48^3	6.28E−01	1.76E+05	1.29E−02	8.55E+06	48.54
96^3	4.56E+00	1.94E+05	7.60E−02	1.16E+07	60.02
144^3	1.40E+01	2.14E+05	2.50E−01	1.20E+07	55.95
192^3	3.18E+01	2.22E+05	5.69E−01	1.24E+07	55.93
O3					
48^3	1.90E+00	5.83E+04	3.27E−02	3.38E+06	58.01
96^3	1.29E+01	6.88E+04	2.01E−01	4.41E+06	64.04
144^3	4.20E+01	7.11E+04	6.38E−01	4.68E+06	65.85
192^3	9.20E+01	7.69E+04	1.46E+00	4.85E+06	63.12
O4					
48^3	7.28E+00	1.52E+04	9.21E−02	1.20E+06	78.98
96^3	4.72E+01	1.88E+04	5.72E−01	1.55E+06	82.44
144^3	1.42E+02	2.10E+04	1.79E+00	1.66E+06	79.20
180^3	2.67E+02	2.18E+04	3.42E+00	1.71E+06	78.23

third order and on CPUs, the ADER can still function in a relatively cache-friendly mode. However, at third order and on GPUs, the ADER is not as efficient because it is not so cache friendly. The fourth-order results in Table 11 are even more interesting. We see that the ADER seems to entirely dominate the time taken for a timestep. This is so on CPUs and GPUs, which tells us that, in either computational environment, the ADER is not so cache friendly. Even though this result is notas optimistic as one would desire, it reveals

Table 11 Comparing the fractional time spent in the ADER subroutine at various orders on CPU and GPU for the CED code

Zones^3	Serial time/s	Fraction of time in ADER	GPU time/s	Fraction of time in ADER
O2				
48^3	3.18E−01	0.506	6.95E−03	0.538
96^3	2.09E+00	0.458	3.56E−02	0.468
144^3	6.63E+00	0.475	1.09E−01	0.437
192^3	1.45E+01	0.457	2.46E−01	0.433
O3				
48^3	1.48E+00	0.778	2.41E−02	0.738
96^3	9.59E+00	0.746	1.49E−01	0.742
144^3	2.95E+01	0.702	4.61E−01	0.723
192^3	6.39E+01	0.694	1.06E+00	0.724
O4				
48^3	6.34E+00	0.871	7.58E−02	0.823
96^3	3.98E+01	0.844	4.71E−01	0.823
144^3	1.18E+02	0.828	1.47E+00	0.817
180^3	2.21E+02	0.826	2.78E+00	0.813

an opportunity. It shows us that there might be room for improving the ADER algorithm at higher orders by finding more cache-friendly variants. It also tells us that machines with larger caches would be more suitable for this type of application.

4 Accuracy Analysis CPU vs. GPU

The astute reader might still point out that the new thing that makes GPUs competitive with CPUs is indeed the inclusion of error correction in the GPUs. Therefore, we are interested in providing some perspective on how good the error correction indeed is on the GPUs. This is done by taking standard test problems that are used for accuracy analysis and running them on a CPU and gathering the results. We then run the same test for exactly the same number of time steps and identical stopping time on a GPU. Subsection 4.1 documents CFD results; Subsect. 4.2 documents MHD results, and Subsect. 4.3 documents results from CED. The happy conclusion from these sections is that the accuracy numbers are virtually identical up to about nine or ten digits of accuracy. As a result, the reader will find the tables that show accuracy below virtually identical for CPU and GPU.

4.1 CFD Results

The result shown is from a three-dimensional version of the isentropic hydrodynamic vortex test problem from Balsara and Shu [13]; see Subsect. 4.2 of that work. Table 12 shows the L_1 and L_∞ errors at second, third, and fourth orders for our CFD test problem. We also show that the schemes on both CPU and GPU do indeed reach their target accuracies. Up to four significant digits, the error from the CPU and GPU look identical. As a result, the last column gives

Table 12 L_1 and L_∞ errors at second, third, and fourth orders for our CFD test problem

CPU vs. GPU O2; Zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	1.60E – 03	1.60E – 03	–	1.98E – 15
96^3	4.03E – 04	4.03E – 04	1.99	6.10E – 17
144^3	1.63E – 04	1.63E – 04	2.23	2.20E – 17
192^3	8.74E – 05	8.74E – 05	2.16	1.31E – 10
CPU vs. GPU O2; Zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	2.82E – 02	2.82E – 02	–	1.23E – 14
96^3	7.47E – 03	7.47E – 03	1.92	1.10E – 16
144^3	3.00E – 03	3.00E – 03	2.25	4.50E – 16
192^3	1.55E – 03	1.55E – 03	2.29	3.10E – 09
CPU vs. GPU O3; Zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	1.09E – 03	1.09E – 03	–	3.90E – 16
96^3	2.25E – 04	2.25E – 04	2.27	9.76E – 19
144^3	7.07E – 05	7.07E – 05	2.85	8.78E – 17
192^3	3.03E – 05	3.03E – 05	2.94	3.20E – 17
CPU vs. GPU O3; Zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	1.84E – 02	1.84E – 02	–	6.00E – 16
96^3	3.21E – 03	3.21E – 03	2.52	5.60E – 16
144^3	1.06E – 03	1.06E – 03	2.74	1.66E – 15
192^3	4.63E – 04	4.63E – 04	2.87	3.33E – 16
CPU vs. GPU O4; zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	1.03E – 03	1.03E – 03	–	1.20E – 14
96^3	9.46E – 05	9.46E – 05	3.44	8.48E – 16
144^3	1.84E – 05	1.84E – 05	4.04	2.07E – 17
192^3	5.48E – 06	5.48E – 06	4.22	2.16E – 17
CPU vs. GPU O4; zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	6.50E – 02	6.50E – 02	–	5.35E – 13
96^3	1.15E – 02	1.15E – 02	2.50	1.21E – 14
144^3	9.64E – 04	9.64E – 04	6.11	1.33E – 15
192^3	4.38E – 04	4.38E – 04	2.74	4.44E – 16

We show the error in the density variable for the hydrodynamical vortex. Up to four significant digits, the error from the CPU and GPU looks identical. As a result, the last column gives the absolute value of the difference between the error as measured on the CPU and the GPU

the absolute value of the difference between the error as measured on the CPU and the GPU. This drives home the point that the difference between the CPU result and the result from the GPU is indeed in the digits that would be considered insignificant for scientific computation. The table shows that on a CPU, as well as on a GPU, virtually identical accuracies are obtained.

Table 13 L_1 and L_∞ errors at second, third, and fourth orders for our MHD test problem

CPU vs. GPU O2; Zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	4.65E – 03	4.65E – 03	–	1.04E – 17
96^3	1.24E – 03	1.24E – 03	1.91	9.97E – 18
144^3	5.58E – 04	5.58E – 04	1.97	4.99E – 18
192^3	3.15E – 04	3.15E – 04	1.99	6.60E – 17
CPU vs. GPU O2; Zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	6.18E – 02	6.18E – 02	–	1.66E – 14
96^3	1.54E – 02	1.54E – 02	2.01	3.30E – 15
144^3	6.83E – 03	6.83E – 03	2.00	1.19E – 15
192^3	3.85E – 03	3.85E – 03	1.99	4.22E – 15
CPU vs. GPU O3; zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	1.43E – 03	1.43E – 03	–	1.18E – 15
96^3	1.92E – 04	1.92E – 04	2.90	8.20E – 17
144^3	5.73E – 05	5.73E – 05	2.98	1.68E – 17
192^3	2.42E – 05	2.42E – 05	2.99	3.99E – 17
CPU vs. GPU O3; zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	2.46E – 02	2.46E – 02	–	1.50E – 15
96^3	3.01E – 03	3.01E – 03	3.03	3.90E – 16
144^3	8.32E – 04	8.32E – 04	3.17	2.16E – 15
192^3	3.59E – 04	3.59E – 04	2.92	8.33E – 16
CPU vs. GPU O4; zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
48^3	1.40E – 03	1.40E – 03	–	7.56E – 14
96^3	6.16E – 05	6.16E – 05	4.51	5.69E – 18
144^3	1.02E – 05	1.02E – 05	4.43	2.31E – 17
192^3	3.00E – 06	3.00E – 06	4.26	1.98E – 17
CPU vs. GPU O4; Zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
48^3	9.32E – 02	9.32E – 02	–	1.03E – 12
96^3	4.45E – 03	4.45E – 03	4.39	3.33E – 15
144^3	6.57E – 04	6.57E – 04	4.72	1.67E – 15
192^3	1.72E – 04	1.72E – 04	4.66	6.33E – 15

We show the error in the x-component of the magnetic variable for the MHD vortex. The results are presented in the same style as in Table 12

4.2 MHD Results

The result shown is from a three-dimensional version of the MHD vortex test problem from Balsara [7]; see Sect. 6 of that work. Table 13 shows the L_1 and L_∞ errors at second, third, and fourth orders for our MHD test problem. The results are presented in the same style as in Table 12. The table shows that on a CPU, as well as on a GPU, virtually identical accuracies are obtained.

Table 14 L_1 and L_∞ errors at second, third, and fourth orders for our CED test problem

CPU vs. GPU O2; Zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
16^3	1.03E – 04	1.03E – 04	–	1.99E – 18
32^3	2.35E – 05	2.35E – 05	2.13	0.00E + 00
64^3	5.68E – 06	5.68E – 06	2.05	2.64E – 18
128^3	1.41E – 06	1.41E – 06	2.01	1.90E – 18
CPU vs. GPU O2; zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
16^3	1.63E – 04	1.63E – 04	–	0.00E + 00
32^3	3.67E – 05	3.67E – 05	2.15	2.00E – 18
64^3	8.92E – 06	8.92E – 06	2.04	2.65E – 18
128^3	2.21E – 06	2.21E – 06	2.01	3.14E – 18
CPU vs. GPU O3; zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
16^3	3.59E – 05	3.59E – 05	–	0.00E + 00
32^3	4.39E – 06	4.39E – 06	3.03	3.05E – 20
64^3	5.41E – 07	5.41E – 07	3.02	1.11E – 19
128^3	6.71E – 08	6.71E – 08	3.01	2.33E – 19
CPU vs. GPU O3; zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
16^3	5.47E – 05	5.47E – 05	–	1.50E – 18
32^3	6.78E – 06	6.78E – 06	3.01	0.00E + 00
64^3	8.43E – 07	8.43E – 07	3.01	1.09E – 18
128^3	1.05E – 07	1.05E – 07	3.01	4.34E – 18
CPU vs. GPU O4; zones	L_1 error CPU	L_1 error GPU	L_1 accuracy	|CPU L_1 error – GPU L_1 error|
16^3	2.04E – 06	2.04E – 06	–	3.90E – 19
32^3	1.14E – 07	1.14E – 07	4.16	6.90E – 20
64^3	7.09E – 09	7.09E – 09	4.01	2.43E – 19
128^3	4.52E – 10	4.52E – 10	3.97	6.04E – 19
CPU vs. GPU O4; zones	L_∞ error CPU	L_∞ error GPU	L_∞ accuracy	|CPU L_∞ error – GPU L_∞ error|
16^3	3.39E – 06	3.39E – 06	–	2.60E – 18
32^3	1.71E – 07	1.71E – 07	4.31	2.28E – 18
64^3	1.12E – 08	1.12E – 08	3.93	2.17E – 18
128^3	7.05E – 10	7.05E – 10	3.99	1.36E – 20

We show the error in the y-component of the electric displacement variable for the electromagnetic wave. The results are presented in the same style as in Table 12

4.3 CED Results

The result shown is from a three-dimensional version of the electromagnetic wave propagation test problem from Balsara et al. [16]; see Subsect. 5.1 of that work. Table 14 shows the L_1 and L_∞ errors at second, third, and fourth orders for our CED test problem. The results are presented in the same style as in Table 12. The table shows that on a CPU, as well as on a GPU, virtually identical accuracies are obtained.

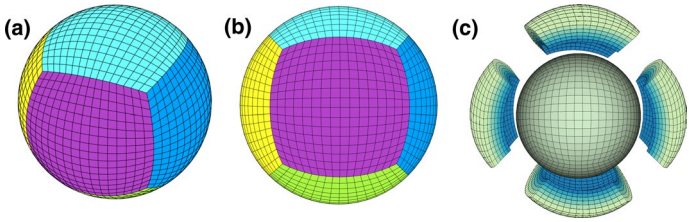

Fig. 8 Meshing of an actual physical problem. A two-dimensional cube sphere mesh is constructed on the surface of a unit sphere in Fig. 8a, and Fig. 8b shows us a rotated version of the same. The two-dimensional surface mesh in Figs. 8a and b results in six sectors, and five of those sectors which face the reader are shown with different colors. The problem consists of doing three-dimensional MHD calculations on the exterior volume of the sphere. To that end, the two-dimensional surface mesh is extended in the third, i.e., radial, direction. This is shown in Fig. 8c. Figure 8c only shows the three-dimensional mesh associated with four of the possible six sectors. The resulting mesh gives us a uniform coverage of the three-dimensional volume that is exterior to the sphere and is known as a three-dimensional cube sphere mesh

5 Scalability Study with Comparable Numbers of GPUs and CPUs

Section 2 has shown us the different types of techniques, tricks, and algorithms needed for extracting performance from a GPU, and Sect. 3 has shown us that the ideas that we have developed indeed work in idealized situations. We now show how well these ideas work in an end-to-end scientific application. It is very illustrative to show a real-world application problem and carry out a scalability study involving GPUs and comparable number of multicore CPUs for such a problem. The problem we use is from astrophysical applications, see Subramanian et al. [36]. The only difference between the cited research and the current application is that now we wish to do the problem with more physics, which calls for a larger mesh and a more sophisticated mesh configuration. Figure 8 shows us the meshing of an actual problem. Of course the number of zones have been downsampled in the figure to make it easier for the reader to visualize the geometry of the problem. Here, the physical problem that we want to solve consists of simulating fully three-dimensional MHD in the volume that is exterior to a sphere, as shown in Fig. 8. Figures 8a, and b show us different views of the two-dimensional surface of a unit sphere, showing us that the surface can be meshed with six sectors. The five sectors that face the reader are shown with different colors; the sixth sector is hidden from view. Figure 8c shows us how the two-dimensional sectors can be converted into three-dimensional sectors by extending the angular mesh from the spherical surface in the radial direction (i.e., the third direction). Only four of the possible six three-dimensional sectors are shown in Fig. 8c. In Fig. 8, we show only a few zones in each direction, but an actual computation involves hundreds of zones in each direction of a three-dimensional sector. Such a three-dimensional mesh that covers the volume that is exterior to a sphere is known as a cube sphere mesh. The goal is to carry out a three-dimensional MHD simulation on such a cube sphere mesh. The scalability will be dictated by the real-world physics application that is to be done on a GPU-rich machine.

The physics of the problem is as follows. We need to cover the sphere with an angular resolution of less than one degree. To that end, we decided that each sector should have 100×100 zones in each of the two angular directions on the unit sphere. Furthermore, in the radial direction, the physics of the problem is such that we need 400 zones in the radial

direction. Therefore, each sector of the type shown in Fig. 8c should have $100 \times 100 \times 400$ zones; and please realize that we have six such sectors in Fig. 8c. On evaluating the application requirements, it became evident to us that all the data associated with $100 \times 100 \times 400$ zones would not fit within the RAM memory of a single GPU. Therefore, the data would have to be chunked into smaller patches and those patches would have to be sent to the device for update and the minimal updated data would have to be brought back to the host. We found that $100 \times 100 \times 100$ zone patches could fit on each of the GPUs that we had access to. As a result, we would have to start our scalability study with 4 such patches per sector and 6 sectors, resulting in 24 patches (each patch having $100 \times 100 \times 100$ zones) for the full spherical calculation. We also found that the host could accommodate eight patches on each of the nodes of the supercomputer we were using. This paragraph serves to show us that the hardware limitations, and the physics of the problem, strongly limit how a calculation can be done on a GPU-rich supercomputer. We discuss the details of the machine that we were using next.

We used the Delta machine at NCSA along with the NVIDIA NVHPC (version 22.2) Fortran compiler. Delta has various configurations. The configuration that we had access to had 100 nodes. Each node had one AMD Milan 64 core CPU (the host) and four A100 GPUs with 40 GB HBM2 RAM and NVLink (the devices). We wished to compare the performance of a certain number of GPUs with the same number of multicore CPUs. As a result, given the size of the physical problem, it was felt that we could do a weak scalability study that went from 12 to 96 GPUs and the same could be compared with a comparable scalability study that went from 12 to 96 CPUs. Each node of Delta had sufficient main RAM to hold 8 patches with $100 \times 100 \times 100$ zones per patch. Delta also has only 100 nodes, each with 1 CPU, so the scalability study would be restricted to a maximum of 96 CPUs, and therefore 96 GPUs.

The application demands high angular resolution. As a result, for the 12 GPU run, we used 100^3 zones per patch and 24 patches. On the CPUs, we have two options as follows.

i) This first option consists of using only one-sided MPI-3 messaging. To carry out the identical run on 12 CPUs (with 64 cores per CPU), we had to use $25 \times 25 \times 50$ zones per patch and 768 patches. Please check that the number of zones is the same; i.e., 24×100^3 zones on the 12 GPUs is same as $25 \times 25 \times 50 \times 12 \times 64$ zones on the 12 CPUs with 64 cores per CPU. With each doubling of CPUs or GPUs, we just doubled the number of patches. One-sided MPI-3 communication was used for the messaging across patches to have a messaging strategy with the lowest latency and highest bandwidth. (Please see Garain et al. [25] to realize that the MPI-3 messaging strategy is optimal and that the on-processor performance of the code is also optimal for this category of code.) All the MPI communication had to be done through the CPUs because the GPUs never had enough RAM to concurrently hold all the data for the whole problem.

ii) The second option consists of using a hybrid parallelization model on the CPUs based on using 64-way OpenMP-based parallelization within a CPU and MPI-3 parallelization across CPUs. This strategy allows us to maintain identical patch sizes on GPUs and CPUs. The CPU patches now had $100 \times 100 \times 100$ zones; i.e., the same as on the GPUs. The benefit of this strategy is that it allows much fewer MPI messages to be exchanged, with each message being larger. The downside of this strategy is that

Table 15 Scalability data when all the computations were done on CPUs with MPI-3 messaging between CPUs

MPI Only	64 Core processors	Zones	CPU time	CPU zones/s
	12	24 000 000	2.45E + 00	9.79E + 06
	24	48 000 000	3.78E + 00	1.27E + 07
	48	96 000 000	6.46E + 00	1.49E + 07
	96	192 000 000	1.14E + 01	1.68E + 07

Table 16 Scalability data when all the computations were done on CPUs with a hybrid OpenMP + MPI-3 parallelization strategy

OpenMP + MPI Hybrid	64 Core processors	Zones	CPU time	CPU zones/s
	12	24 000 000	3.25E + 00	7.39E + 06
	24	48 000 000	3.62E + 00	1.33E + 07
	48	96 000 000	3.81E + 00	2.52E + 07
	96	192 000 000	3.90E + 00	4.92E + 07

Table 17 Scalability data when all the computations were done on GPUs with MPI-3 messaging between CPUs

A100 GPUs	Zones	GPU time	GPU zones/s	GPU vs. CPU speed-up Only MPI	GPU vs. CPU speed-up OpenMP + MPI
12	24 000 000	5.87E − 01	4.09E + 07	4.17	5.53
24	48 000 000	6.24E − 01	7.70E + 07	6.07	5.80
48	96 000 000	6.70E − 01	1.43E + 08	9.63	5.69
96	192 000 000	7.75E − 01	2.48E + 08	14.70	5.03

The speedups on the GPUs relative to the CPU-based information in Tables 15 and 16 are also shown

one has to include OpenMP parallelization in addition to MPI parallelization. In this approach too, all the MPI communication had to be done through the CPUs because the GPUs never had enough RAM to concurrently hold all the data for the whole problem.

(i) The discussion in this paragraph is intended to bring the reader face to face with the harsh realities and substantial limitations that a computationalist faces when simulating a large real-word scientific application on an actual GPU-equipped supercomputer!

(ii) Table 15 shows the scalability study when only MPI parallelization was used on the CPUs. Table 16 shows the same scalability when a hybrid OpenMP + MPI parallelization paradigm was used on the CPUs; we see that the results in Table 16 are a nice improvement over Table 15. Table 17 shows the scalability study when only MPI was used across the CPUs but all the computation was done on GPUs. (As has been explained before, the structure of the application was such that it was not possible to use MPI natively on the GPUs.) The speed advantages of the GPUs relative

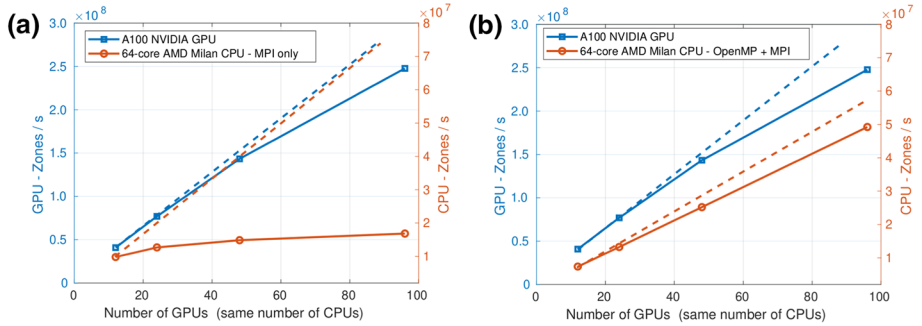

Fig. 9 Scalability study for the numerical code illustrated in Fig. 8. The scalability study spans 12–96 GPUs. To make this a fair comparison with CPUs, we conducted identical scalability studies that span 12–96 CPUs. The solid blue curves in Figs. 9a and b show the actual GPU results with MPI-3 messaging. The dashed blue line shows ideal scalability. In Fig. 9a, the solid red curve shows the actual CPU results with MPI-3 messaging. Because only MPI was used, the patch sizes were much smaller. The dashed red line shows the ideal scalability when using MPI-3 messaging. In Fig. 9b, the solid red curve shows the actual CPU results when a hybrid OpenMP + MPI-3 parallelization model was used. Because we used OpenMP within the CPU and MPI across CPUs, the patch sizes were much larger. The dashed red line in Fig. 9b shows the ideal scalability when the hybrid OpenMP + MPI-3 parallelization model was used. Please note that GPU data should be read off the left vertical axis while CPU data should be read off the right vertical axis

to the two CPU approaches (with different parallelization strategies) are shown in the last two columns of Table 17. From the second to last column of Table 17, we see that using a parallelization model on CPUs that is based exclusively on MPI is not a winning strategy because the CPU performance degrades with increasing numbers of CPUs. This is because the patches on the CPUs are small and a lot of ghost zone information has to be exchanged via a lot of messages. However, from the last column in Table 17, we see that when a hybrid OpenMP + MPI parallelization strategy is followed for CPU computing, then the patches can be made larger on the CPUs and the number of messages exchanged can be substantially reduced. From the last column of Table 17, we see that if patches of the same size are used for GPU computing and for CPU computing, then the GPUs are about five to six times faster than a comparable number of high-end multicore CPUs. Moreover, Table 17 shows this speed advantage is sustained over a substantial range in the number of GPUs, or CPUs, that are used for the scalability study.

(iii) Figures 9a and b show plots of the information shown in Tables 15, 16, and 17. The blue curves in Figs. 9a and b show the zones per second updated with the GPU-based runs as a function of the number of GPUs used. Because the GPU runs are so much faster than the CPU runs, please use the left vertical axis when viewing GPU data. The red curve in Fig. 9a shows the zones per second updated as a function of the number of CPUs used when an MPI only model was used for the parallelism on the CPUs. The red curve in Fig. 9b shows the zones per second updated as a function of CPUs used when a hybrid OpenMP-MPI model was used for the parallelism on the

CPUs. Because the CPU runs were so much slower than the GPU runs, please use the right vertical axis when viewing CPU data in Fig. 9. Figure 9 brings the advantage of GPU computing, as described in this paper, into a very sharp focus!

6 Conclusions

We started with the important realization that GPU computing will be an integral part of any Exascale supercomputer in the foreseeable future. Since a variety of higher order Godunov (HOG) schemes are expected to constitute about 30% of the application mix on such supercomputers, it is important to explore how such HOG schemes can be ported to such advanced supercomputer architectures. To that end, we identify three representative areas—CFD, MHD, and CED where HOG schemes are used in quite different styles. We realize that if a unified pathway can be shown where all such applications can be ported (without too much effort) to such machines, that would enhance both the value of HOG schemes and their utilization of GPU-equipped supercomputers.

To help in the overarching goal of making HOG schemes ready for GPU architectures, we found that OpenACC is a good vehicle for expressing the parallelism. OpenACC is a language extension, with the result that a rich set of pragmas enable us to give advice to the GPU on how to parallelize code. We also acknowledge that successful algorithms should be able to operate efficiently within the confines of the very small caches that are available on GPUs. The interconnect between CPU (host) and GPU (device) is also rather slow with the result that tricks have to be devised to ensure that there is minimal data motion.

After factoring in the computational landscape within which a HOG scheme has to operate on Exascale supercomputers, we came to the realization that not every algorithm that is known to the applied math community will thrive in the constricted environment of GPU computing. Therefore, it was necessary to identify successful algorithmic choices. WENO schemes were found to be very favorable approaches for starting with a minimal amount of data on the GPU and carrying out all of the spatial reconstruction at higher order. Furthermore, WENO reconstruction can be written in a way so as to optimally work with the very small caches on GPUs. To avoid frequent data exchanges between host and node and vice versa, ADER predictor steps were found to be a very favorable way of starting with the spatial reconstruction and obtaining from it a space-time higher order representation of the PDE. The Riemann-solver based corrector step had also to be examined, and only certain Riemann solvers (like HLL), which are parsimonious in their use of caches, were found to be favorable.

Having identified the optimal choices, it is important to show the community how they can be implemented using the common programming languages that are currently employed. Therefore, Sect. 2 of this paper shows how all parts of a HOG scheme for CFD are easily implemented using OpenACC extensions to Fortran code. The Appendix does the same for C/C++code. With that, this paper serves as a one-stop shop for giving the reader advice on all the essential OpenACC pragmas that s/he will need to make a high-quality OpenACC implementation.

Just a naïve parallelization with OpenACC will not unlock the dramatic performance gains that GPUs potentially offer. The most important trick that we found was to identify “skinny”

variables that move the absolute minimum data from host to node and vice versa. Without this trick, Table 3 shows that the GPU speedup invariably suffers. Furthermore, it degrades with increasing order of accuracy. When this trick is properly used, Table 4 shows that when we go from second to third order, we actually see a performance gain. This is because we are able to utilize GPU cores much more efficiently to carry out the floating point intensive operations of higher order schemes. GPUs do indeed have very tiny caches (compared to CPU caches). As a result, Table 5 identifies a "Predictor bottleneck". In other words, while present generation WENO schemes and Riemann solvers can function efficiently on very small caches, the ADER predictor algorithm still cannot work very efficiently with the very small caches on a GPU. We do prove, however, that ADER can be written so that it functions very efficiently with the larger caches on a CPU. In that sense, the paper also serves to identify the improvements thata future GPU architectures and future algorithmic designs can make to close this gap.

We have also realized that not everyone might want to go through the intricacies of implementing an ADER predictor step. For that reason, we also explored the performance of Runge-Kutta time stepping for CFD and MHD on GPUs. Since this form of time stepping bypasses the ADER step, we have been able to show that the speedup for third-order Runge-Kutta-based schemes is better than the speedup for second-order Runge-Kutta-based schemes. Likewise, the speedup for fourth-order Runge-Kutta-based schemes is better than the speedup for third-order Runge-Kutta-based schemes. Overall, however, the ADER time stepping is faster than the Runge-Kutta time stepping on both CPUs and GPUs. Section 5 ties all the previous sections together by describing the process of carrying out a scalability study on a GPU-rich supercomputer and comparing it to an analogous CPU-based scalability study. This section is very useful because it shows us how substantial the gains can be if GPUs are mastered, but it also shows us how one has to dexterously circumvent some of the limitations of GPUs.

The paper began by saying that GPU computing offered gigantic potential gains, if such gains could be realized. The results obtained show that with the right algorithmic choices, the right OpenACC directives, and the right tricks, these gains can be mostly realized for a broad range of HOG schemes. This is especially true for larger meshes where the A100 GPU shows better than 100x speedup relative to a single core of a CPU. This speedup is certainly much better than the typical speedups that are often quoted. Indeed, these fantastic gains are only realized if we pay careful attention to all aspects of the algorithm as well as all the limitations of present-day GPU architectures. It is, however, very satisfying that such gains can indeed be realized.

GPU computing is an evolving enterprise. As other vendors see the commercial benefits of GPUs, they will design better GPUs that are more dedicated to HPC and knowledge discovery. All the results here are based on the A100 GPU. The A100 GPU evolved from the V100 GPU and had several improvements in bandwidth and processing power compared to its predecessor. It is inevitable that the A100 GPU will give way to even better GPU products. While we do not have access to the H100 GPU from NVIDIA, early reports seem to indicate that it has vastly improved bandwidth to main memory. Such improvements will make it possible to find efficient solution strategies for some of the other algorithms mentioned here, such as the DG methods and their variants. GPU computing is, after all, an evolving enterprise. The attention to algorithms that is given in the early sections of this paper is intended to be evolutionary and advisory in spirit. As new GPU architectures come online, we will find more opportunities to bring other higher order algorithms into the fold of GPU computing. The deep analysis of OpenACC and its usage in these algorithms will, nevertheless, be of value to all such algorithms.

Appendix A

The OpenACC implementation of a HOG scheme in the C/C++ language is described here. The overall structure is identical to that of the Fortran code structure described in Sect. 2.2. The reader who is familiar with C/C++ should be able to pick up all the OpenACC insights from the figures shown in this Appendix. In this Appendix, we also highlight the differences between Fortran and C/C++ syntax and use of the OpenACC directives.

Figure A1 shows the header file containing the preprocessor definitions of the domain sizes "*nx*, *ny*, *nz*" and the function declarations. The data type and array sizes are specified in the function interface as required by the C/C++ syntax, which is different from Fortran. It can be noted that the variable "d*t*_next" in the function "Update()" is passed by reference as a pointer since this is a scalar output variable. Such special care is not needed in a Fortran subroutine,

```
#include<stdio.h>
#include<math.h>

#define nx 50
#define ny 50
#define nz 50

// Function declarations

int Initialize(double U[nz+4][ny+4][nx+4][5][5]);
int U_Skinny_to_U (double U_skinny[nz+4][ny+4][nx+4][5],
                   double U[nz+4][ny+4][nx+4][5][5]);
int Boundary_Conditions(double U[nz+4][ny+4][nx+4][5][5]);
int Limiter            (double U[nz+4][ny+4][nx+4][5][5]);
int Predictor          (double U[nz+4][ny+4][nx+4][5][5]);
int Make_Flux_X(double U[nz+4][ny+4][nx+4][5][5],
                double Flux_X[nz+4][ny+4][nx+5][5]);
int Make_Flux_Y(double U[nz+4][ny+4][nx+4][5][5],
                double Flux_Y[nz+4][ny+5][nx+4][5]);
int Make_Flux_Z(double U[nz+4][ny+4][nx+4][5][5],
                double Flux_Z[nz+5][ny+4][nx+4][5]);
int Make_dU_dt (double Flux_X[nz+4][ny+4][nx+5][5],
                double Flux_Y[nz+4][ny+5][nx+4][5],
                double Flux_Z[nz+5][ny+4][nx+4][5],
                double dU_dt[nz+4][ny+4][nx+4][5],
                double dt, double dx, double dy, double dz);
int Update(double U[nz+4][ny+4][nx+4][5][5],
           double U_skinny[nz+4][ny+4][nx+4][5],
           double dU_dt[nz+4][ny+4][nx+4][5],
           double *dt_next, double cfl, double dx, double dy, double dz);
```

Fig. A1 Header file "hydro.h", containing the definitions of the domain sizes "*nx*, *ny*, and *nz*" in the *x*-, *y*-, and *z*-directions, respectively; and also the function declarations, which are essential for a hydrodynamic simulation. This header file is common to all the C/C++ functions and the main function

```
#include "hydro.h"
int main()
{
    double U[nz+4][ny+4][nx+4][5][5], U_skinny[nz+4][ny+4][nx+4][5],
       dU_dt[nz+4][ny+4][nx+4][5], Flux_X[nz+4][ny+4][nx+5][5],
      Flux_Y[nz+4][ny+5][nx+4][5], Flux_Z[nz+5][ny+4][nx+4][5];
    double dt, dt_next, cfl, dx, dy, dz;
    const int n_timesteps = 500;
    #pragma acc data create(U_skinny, U, dU_dt, \
        Flux_X, Flux_Y, Flux_Z, dt, dt_next, cfl, dx, dy, dz)

    Initialize(U_skinny, dt, cfl, dx, dy, dz);   // Executed on host (CPU)
    #pragma update device(U_skinny, dt, cfl, dx, dy, dz)

    for(int it = 0; it < n_timesteps; it++){
        #pragma acc update device(U_skinny, dt)
        U_Skinny_to_U(U_skinny, U);
        Boundary_Conditions(U);
        Limiter(U);
        Predictor(U);
        Make_Flux_X(U, Flux_X);
        Make_Flux_Y(U, Flux_Y);
        Make_Flux_Z(U, Flux_Z);
        Make_dU_dt (Flux_X, Flux_Y, Flux_Z, dU_dt, dt, dx, dy, dz);
        Update_U_Timestep(U, U_skinny, dU_dt, &dt_next, cfl, dx, dy, dz);
        #pragma acc update host(U_skinny)
        dt = dt_next;
    }
    return 0;
}
```

Fig. A2 Structure of a C/C++ version of the hydro code (in pseudocode format) at second order with OpenACC extensions to allow it to perform well on a GPU. The overall structure is identical to that of the Fortran version. The exceptions are, the arrays dimensions starts from "0" in C and the zone indexing is reversed from "(i, j, k)" of Fortran to "[k][j][i]" of C, to efficiently operate on the row-major array storage format in C. The OpenACC pragmas are identical to the Fortran version, except the C/C++ version does not need the ending "#pragma acc end" pragma

since all the variables are passed by reference by default. Similar to the Fortran code, the function "U_skinny_to_U()" moves the minimal data from CPU to GPU at the beginning of every timestep.

Figure A2 shows the overall structure of a hydro code in C/C++, which is analogous to the Fortran structure shown in Fig. 1. The OpenACC directives have a different structure in C/C++, they start with "#pragma acc" and they do not require an "!$acc end" statement. So, the analogous form for "!$acc end data" in Fig. 1 is not present in Fig. 2.

```
#include "hydro.h"
int U_Skinny_to_U(double U_skinny[nz+4][ny+4][nx+4][5],
                  double U[nz+4][ny+4][nx+4][5][5])
{
    int i, j, k, n;

    #pragma acc parallel vector_length(64) present(U_skinny, U)
    #pragma acc loop gang vector collapse(3) independent \
            private(i, j, k, n)
    for(k = 0; k < nz+4; k++){
        for(j = 0; j < ny+4; j++){
            for(i = 0; i < nx+4; i++){

                #pragma acc loop seq
                for(n = 0; n < 5; n++){
                    U[k][j][i][n][0] = U_skinny[k][j][i][n];
                }
            }
        }
    }
    return 0;
}
```

Fig. A3 Structure of the "*U*_Skinny_to_*U*" function. The array "*U*_skinny" contains the minimum data that should be carried from the host to the device. This subroutine just copies the contents of "*U*_skinny" to "*U*(:, :, :, :, 1)". The OpenACC pragmas are identical to that of the Fortran version

Figure A3 shows the C/C++ code for the "*U*_skinny_to_*U*()" function. This is equivalent to the Fortran code shown in Fig. 2. The OpenACC directives that are outside the triply nested loop are identical in C/C++ and Fortran. The Fortran has the ":" operator for the whole array operation, whereas the C/C++ does not have any special operator for this. Therefore, we need another "for" loop inside the triply nested loop. This inner for-loop will be executed serially. It has been marked with the OpenACC directive "#pragma acc loop seq".

Figure A4 shows the overall structure of a "Limiter()" function in C/C++. This is analogous to the Fortran subroutine shown in Fig. 3. The OpenACC directives are identical, except for the "!$acc end" directive, which is not needed in C/C++. Unlike Fortran, the C/C++ does not allow functions to be defined inside another function. So, the "MC_limiter()" function is defined as an inline function, outside the "Limiter()". The "MC_limiter()" is called inside a triply nested loop serially. So, it is marked with "#pragma acc routine seq".

Figure A5 shows the overall structure of a "Predictor()" function in C/C++. This is equivalent to the Fortran subroutine, shown in Fig. 4. The overall structure and the OpenACC directives are identical to the Fortran code. As mentioned before, the C/C++ lacks the ":" operator for array operations. Due to this, "for" loops are used for the array operations, inside the triply nested loop. These are marked with "#pragma acc loop seq" to indicate their serial execution.

Figure A6 shows the C/C++ code for the "Make_Flux()" function. This is equivalent to the Fortran code shown in Fig. 5. The OpenACC directives are identical to the Fortran code. An

```
#include "hydro.h"
#pragma acc routine seq
static inline double MC_Limiter(double a, double b, double mc_coef){
    // Build and return the limited value
}
int Limiter(double U[nz+4][ny+4][nx+4][5][5])
{
    int i, j, k, n;    double compression_factor, a, b;
    #pragma acc data copyin(n, compression_factor)
    for(n = 0; n < 5; n++){
        if(n == 0) compression_factor = 2.0;
        else compression_factor = 1.5;
        #pragma acc update device(n, compression_factor)
        #pragma acc parallel vector_length(64) present(n, U, \
                compression_factor)
        #pragma acc loop gang vector collapse(3) independent \
                private(i, j, k, a, b)
        for(k = 1; k < nz+3; k++){
            for(j = 1; j < ny+3; j++){
                for(i = 1; i < nx+3; i++){
                    a = + U[k][j][i+1][n][0] - U[k][j][i][n][0];
                    b = - U[k][j][i-1][n][0] + U[k][j][i][n][0];
                    U[k][j][i][n][1] = MC_Limiter(a, b, compression_factor);
                }
            }
        }
    }
    // Do similarly for other modes in the y- and z-directions
    return 0;
}
```

Fig. A4 C/C++ version of the application of a simple monotone-centered limiter in the x-direction to all the fluid variables. The inline function "MC_Limiter" is made to accommodate a compression factor which can indeed be changed based on the flow field being limited. The OpenACC directives are identical to that of the Fortran code, except the C version does not need the ending "#pragma acc end" pragma. Also, the inline function must be identified as "acc routine seq" pragma, since the C does not differentiate between a function and a subroutine like Fortran does

```
#include "hydro.h"
#pragma acc routine seq
int Predictor_Zonewise(double U_zonewise[5][5]){
    // Zone-wise predictor step
}
int Predictor(double U[nz+4][ny+4][nx+4][5][5])
{
    int i, j, k, n, m;    double U_zonewise[5][5];
    #pragma acc routine(Predictor_Zonewise) seq

    #pragma acc parallel vector_length(64) present(U)
    #pragma acc loop gang vector collapse(3) independent \
            private(i, j, k, n, m, U_zonewise)
    for(k = 1; k < nz+3; k++){
        for(j = 1; j < ny+3; j++){
            for(i = 1; i < nx+3; i++){
                #pragma acc loop seq
                for(n = 0; n < 5; n++)
                    for(m = 0; m < 4; m++)
                        U_zonewise[n][m] = U[k][j][i][n][m];
                Predictor_Zonewise(U_zonewise);
                #pragma acc loop seq
                for(n = 0; n < 5; n++)
                    U[k][j][i][n][4] = U_zonewise[n][4];
            }
        }
    }
    return 0;
}
```

Fig. A5 C/C++ version of the predictor step for building the fifth mode, i.e., the mode that contains the time rate of change. The OpenACC directives are identical to that of the Fortran code, except in the C version the inner for-loops need to be marked with the "acc loop seq" pragma indicating their sequential execution. This is not needed in the Fortran version, where the ":" operator is used for such operations

```
#include "hydro.h"
#pragma acc routine seq
int Riemann_Solver_Ptwise(double U_L[5], double U_R[5], double flux_x_ptwise[5]){
    // Build the HLL/HLLEM flux
}
int Make_Flux_X(double U[nz+4][ny+4][nx+4][5][5],
           double Flux_X[nz+4][ny+4][nx+5][5])
{
    int i, j, k, n;     double U_L[5], U_R[5], flux_x_ptwise[5];
    #pragma acc routine(Riemann_Solver_Ptwise) seq

    #pragma acc parallel vector_length(64) present(U, Flux_X)
    #pragma acc loop gang vector collapse(3) independent \
            private(i, j, k, n, U_L, U_R, flux_x_ptwise)
    for(k = 2; k < nz+2; k++){
        for(j = 2; j < ny+2; j++){
            for(i = 2; i < nx+3; i++){
                #pragma acc loop seq
                for(n = 0; n < 5; n++){
                    U_L[n] = U[k][j][i-1][n][0] + 0.5*U[k][j][i-1][n][1] \
                                                + 0.5*U[k][j][i-1][n][4];
                    U_R[n] = U[k][j][i][n][0] - 0.5*U[k][j][i][n][1] \
                                              + 0.5*U[k][j][i][n][4];
                }
                Riemann_Solver_Ptwise(U_L, U_R, flux_x_ptwise);
                #pragma acc loop seq
                for(n = 0; n < 5; n++)
                    Flux_X[k][j][i][n] = flux_x_ptwise[n];
            }
        }
    }
    return 0;
}
```

Fig. A6 C/C++ version of the pseudocode for the subroutine to build the flux along the x-faces. The "U_L" variable stores the value of "U" obtained from the zone which is at the left to the face and the "U_R" variable stores the value of "U" obtained from the zone which is at the right to the face. Thus, the facial values of the conserved variable "U" are stored in "U_L, U_R". This is then fed into the Riemann solver function to build the flux for the corresponding face. The OpenACC directives are identical to that of the Fortran code, except in the C version the inner for-loops need to be marked with the "acc loop seq" pragma indicating their sequential execution. This is not needed in the Fortran version, where the ":" operator is used for such operations. In addition, it can be noted that the index of small arrays such as "U_L, U_R, flux_x_ptwise" starts with "0" instead of "1"

additional "#pragma acc loop seq" is used for the inner loop, which performs the operations on each fluid component.

Figure A7 shows the equivalent C/C++code for the "Make_dU_dt()" function. This is analogous to the Fortran shown in Fig. 6. The overall structure and the OpenACC directives are identical between the C/C++and Fortran.

```
#include "hydro.h"
int Make_dU_dt (double Flux_X[nz+4][ny+4][nx+5][5],
                double Flux_Y[nz+4][ny+5][nx+4][5],
                double Flux_Z[nz+5][ny+4][nx+4][5],
                double  dU_dt[nz+4][ny+4][nx+4][5],
                double dt, double dx, double dy, double dz)
{
    int i, j, k, n;

    #pragma acc parallel vector_length(64) present(      \
            dt, dx, dy, dz, Flux_X, Flux_Y, Flux_Z, dU_dt)
    #pragma acc loop gang vector collapse(3) independent \
            private(i, j, k, n)
    for(k = 2; k < nz+2; k++){
        for(j = 2; j < ny+2; j++){
            for(i = 2; i < nx+2; i++){
                #pragma acc loop seq
                for(n = 0; n < 5; n++){
                    dU_dt[k][j][i][n] = \
                     - dt*(Flux_X[k][j][i+1][n] - Flux_X[k][j][i][n]) / dx \
                     - dt*(Flux_Y[k][j+1][i][n] - Flux_Y[k][j][i][n]) / dy \
                     - dt*(Flux_Z[k+1][j][i][n] - Flux_Z[k][j][i][n]) / dz;
                }
            }
        }
    }
    return 0;
}
```

Fig. A7 C/C++ version of the function which builds the "dU_dt" update from the fluxes obtained from the "Make_Flux_X, Make_Flux_Y and Make_Flux_Z" functions. The ":" operator of the Fortran is replaced with a sequential inner for-loop

Figure A8 shows the C/C++code for the "Update_U_Timestep()" function. Here we perform the time update for the conserved variable "U" and estimate the next timestep, "dt_next". This is analogous to the Fortran code shown in Fig. 7. In the C/C++version, the scalar variable "dt_next" is passed by reference as a pointer. In such a case, the function and the OpenACC directives are unaware of the data structure of the pointer variable. Therefore, to keep it simple, an additional temporary variable "dt1" is used for the reduction operation. The variable "dt1" is copied to GPU at the beginning, using the directive "#pragma acc data copyin(dt1)". At the end of the triply nested loop, the final data stored in "dt1" is copied back to the CPU, using the OpenACC directive "#pragma acc update host(dt1)". Then, it is copied to the "dt_next" variable.

```
#include "hydro.h"
int Update_U_Timestep(double U[nz+4][ny+4][nx+4][5][5],
                      double U_skinny[nz+4][ny+4][nx+4][5],
                      double dU_dt[nz+4][ny+4][nx+4][5],
                      double *dt_next, double cfl, double dx, double dy, double dz)
{
    int i, j, k, n;      double U_ptwise[5], dt1 = 1.0e32;

    #pragma acc routine(Eval_Tstep_ptwise) seq
    #pragma acc data copyin (dt1)

    #pragma acc parallel vector_length(64) present(      \
            U, U_skinny, dU_dt, cfl, dx, dy, dz)
    #pragma acc loop gang vector collapse(3) independent \
            private(i, j, k, n, U_ptwise) \
            reduction(min:dt1)
    for(k = 2; k < nz+2; k++){
        for(j = 2; j < ny+2; j++){
            for(i = 2; i < nx+2; i++){
                #pragma acc loop seq
                for(n = 0; n < 5; n++){
                    U[k][j][i][n][0] += dU_dt[k][j][i][n];
                    U_skinny[k][j][i][n] = U[k][j][i][n][0];
                    U_ptwise[n] = U[k][j][i][n][0];
                }
              dt1 = fmin(dt1, Eval_Tstep_ptwise(cfl, dx, dy, dz, U_ptwise));
            }
        }
    }
    #pragma acc update host(dt1)
    *dt_next = dt1;
    return 0;
}
```

Fig. A8 C/C++ version of the function which updates the "*U*" from the "d*U*_d*t*". In addition to this, here the "*U*_skinny" is updated from the new zone-centered values of "*U*". The OpenACC directives are identical to that of the Fortran code, except in the C version the inner for-loop need to be marked with the "acc loop seq" pragma indicating its sequential execution. Also in this function, the next timestep "d*t*_next" is estimated, using the CFL number, zone size, and *U*. The smallest among all the zones is found using the "fmin()" function. To correctly parallelize this reduction operation, the "reduction(min: d*t*1)" pragma is used

Acknowledgements DSB acknowledges support via the NSF grants NSF-19-04774, NSF-AST-2009776, NASA-2020-1241 and the NASA grant 80NSSC22K0628. DSB and HK acknowledge support from a Vajra award.

Funding The funding has been acknowledged. DSB acknowledges support via the NSF grants NSF-19–04774, NSF-AST-2009776, NASA-2020–1241 and the NASA grant 80NSSC22K0628.

Data Availability All data generated or analysed during this study are included in this published article.

Compliance with Ethical Standards

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Ethical approval This manuscript complies with all ethical standards for scientific publishing.

Informed Consent Not applicable.

References

1. Balsara, D.S.: A two-dimensional HLLC Riemann solver with applications to Euler and MHD flows. J. Comp. Phys. **231**, 7476–7503 (2012)
2. Balsara, D.S.: Divergence-free adaptive mesh refinement for magnetohydrodynamics. J. Comput. Phys. **174**, 614–648 (2001)
3. Balsara, D.S.: Divergence-free reconstruction of magnetic fields and WENO schemes for magnetohydrodynamics. J. Comput. Phys. **228**, 5040–5056 (2009)
4. Balsara, D.S.: Higher order accurate space-time schemes for computational astrophysics – Part I: finite volume methods. Liv. Rev. Computat. Astrophy. **3**, 2 (2017). https://doi.org/10.1007/s41115-017-0002-8
5. Balsara, D.S.: Multidimensional extension of the HLLE Riemann solver; application to Euler and magnetohydrodynamical flows. J. Comput. Phys. **229**, 1970–1993 (2010)
6. Balsara, D.S.: Multidimensional Riemann problem with self-similar internal structure – Part I - application to hyperbolic conservation laws on structured meshes. J. Comput. Phys. **277**, 163–200 (2014)
7. Balsara, D.S.: Second-order-accurate schemes for magnetohydrodynamics with divergence-free reconstruction. Astrophys. J. Suppl. **151**, 149–184 (2004)
8. Balsara, D.S., Amano, T., Garain, S., Kim, J.: High order accuracy divergence-free scheme for the electrodynamics of relativistic plasmas with multidimensional Riemann solvers. J. Comput. Phys. **318**, 169–200 (2016)
9. Balsara, D.S., Garain, S., Shu, C.-W.: An efficient class of WENO schemes with adaptive order. J. Comput. Phys. **326**, 780–804 (2016)
10. Balsara, D.S., Meyer, C., Dumbser, M., Du, H., Xu, Z.: Efficient implementation of ADER schemes for Euler and magnetohydrodynamical flows on structured meshes – comparison with Runge-Kutta methods. J. Comput. Phys. **235**, 934–969 (2013)
11. Balsara, D.S., Nkonga, B.: Formulating multidimensional Riemann solvers in similarity variables – Part III – a multidimensional analogue of the HLLI Riemann solver for conservative hyperbolic systems. J. Comput. Phys. **346**, 25–48 (2017)
12. Balsara, D.S., Rumpf, T., Dumbser, M., Munz, C.-D., Efficient, high accuracy ADER-WENO schemes for hydrodynamics and divergence-free magnetohydrodynamics. J. Comput. Phys. **228**, 2480–2516 (2009)
13. Balsara, D.S., Shu, C.-W.: Monotonicity preserving weighted non-oscillatory schemes with increasingly high order of accuracy. J. Comput. Phys. **160**, 405–452 (2000)
14. Balsara, D.S., Spicer, D.S.: A staggered mesh algorithm using high order Godunov fluxes to ensure solenoidal magnetic fields in magnetohydrodynamic simulations. J. Comput. Phys. **149**, 270–292 (1999)
15. Balsara, D.S., Taflove, A., Garain, S., Montecinos, G.: Computational electrodynamics in material media with constraint-preservation, multidimensional Riemann solvers and sub-cell resolution – part I, second-order FVTD schemes. J. Comput. Phys. **349**, 604–635 (2017)
16. Balsara, D.S., Taflove, A., Garain, S., Montecinos, G.: Computational electrodynamics in material media with constraint-preservation, multidimensional Riemann solvers and sub-cell resolution – part II, higher-order FVTD schemes. J. Comput. Phys. **354**, 613–645 (2018)
17. Chandrasekaran, S., Juckeland, G.: OpenACC for Programmers: Concepts and Strategies. Addison-Wesley, Boston (2018)
18. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, Cambridge, MA (2008)
19. Colella, P.: Multidimensional upwind methods for hyperbolic conservation laws. J. Comput. Phys. **87**, 171 (1990)
20. Dai, W., Woodward, P.R.: On the divergence-free condition and conservation laws in numerical simulations for supersonic magnetohydrodynamic flows. Astrophys. J. **494**, 317–335 (1998)
21. Dumbser, M., Balsara, D.S.: A new, efficient formulation of the HLLEM riemann solver for general conservative and non-conservative hyperbolic systems. J. Comput. Phys. **304**, 275–319 (2016)

22. Dumbser, M., Balsara, D.S., Toro, E.F., Munz, C.-D.: A unified framework for the construction of one-step finite volume and discontinuous Galerkin schemes on unstructured meshes. J. Comput. Phys. **227**, 8209–8253 (2008)
23. Dumbser, M., Zanotti, O., Hidalgo, A., Balsara, D.S.: ADER-WENO Finite volume schemes with space-time adaptive mesh refinement. J. Comput. Phys. **248**, 257–286 (2013)
24. Einfeldt, B., Munz, C.-D., Roe, P.L., Sjogreen, B.: On Godunov-type methods near low densities. J. Comput. Phys. **92**, 273–295 (1991)
25. Garain, S., Balsara, D.S., Reid, J.: Comparing Coarray Fortran (CAF) with MPI for several structured mesh PDE applications. J. Comput. Phys. **297**, 237–253 (2015)
26. Godunov, S.K.: Finite difference methods for the computation of discontinuous solutions of the Equations of Fluid Dynamics. Mathematics of the USSR, Sbornik. **47**, 271–306 (1959)
27. Harten, A., Engquist, B., Osher, S., Chakravarthy, S.: Uniformly high order essentially non-oscillatory schemes III. J. Comput. Phys. **71**, 231–303 (1987)
28. Harten, A., Lax, P.D., van Leer, B.: On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. SIAM Rev. **25**, 289–315 (1983)
29. Jiang, G.-S., Shu, C.-W.: Efficient implementation of weighted ENO schemes. J. Comput. Phys. **126**, 202–228 (1996)
30. Roe, P.L.: Approximate Riemann solver, parameter vectors and difference schemes. J. Comput. Phys. **43**, 357–372 (1981)
31. Rusanov, V.V.: Calculation of interaction of non-steady shock waves with obstacles. J. Comput. Math. Phys. USSR **1**, 267 (1961)
32. Ryu, D., Miniati, F., Jones, T.W., Frank, A.: A divergence-free upwind code for multidimensional magnetohydrodynamic flows. Astrophys. J. **509**, 244–255 (1998)
33. Shu, C.-W.: Total variation-diminishing time discretizations. SIAM J Sci. Stat. Comput. **9**, 1073–1084 (1988)
34. Shu, C.-W., Osher, S.J.: Efficient implementation of essentially non-oscillatory shock capturing schemes. J. Comput. Phys. **77**, 439–471 (1988)
35. Shu, C.-W., Osher, S.J.: Efficient implementation of essentially non-oscillatory shock capturing schemes II. J. Comput. Phys. **83**, 32–78 (1989)
36. Subramanian, S., Balsara, D.S., Gagne, M., A.: ud-Doula, Modeling magnetic massive stars in 3D i: isothermal simulations of a magnetic O star. Month. Note. Royal Astronom. Soc. **515**(1), 237–255 (2022)
37. Titarev, V.A., Toro, E.F.: ADER: arbitrary high order Godunov approach. J. Sci. Comput. **17**(1/2/3/4), 609–618 (2002)
38. Titarev, V.A., Toro, E.F.: ADER schemes for three-dimensional nonlinear hyperbolic systems. J. Comput. Phys. **204**, 715–736 (2005)
39. Toro, E.F., Spruce, M., Speares, W.: Restoration of contact surface in the HLL Riemann solver. Shock Waves **4**, 25–34 (1994)
40. Toro, E.F., Titarev, V.A.: Solution of the generalized Riemann problem for advection reaction equations. Proc. R. Soc. Lond. Ser. A **458**, 271–281 (2002)
41. Van Leer, B.: Toward the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. J. Comput. Phys. **32**, 101–136 (1979)
42. Woodward, P., Colella, P.: The numerical simulation of two-dimensional fluid flow with strong shocks. J. Comput. Phys. **54**, 115–173 (1984)

Authors and Affiliations

Sethupathy Subramanian[1] · Dinshaw S. Balsara[1,2] · Deepak Bhoriya[1,3] · Harish Kumar[3]

✉ Dinshaw S. Balsara
dbalsara@nd.edu

Sethupathy Subramanian
ssubrama@nd.edu

Deepak Bhoriya
dbhoriy2@nd.edu

Harish Kumar
hkumar@iitd.ac.in

[1] Physics Department, University of Notre Dame, South Bend, IN, USA

[2] ACMS Department, University of Notre Dame, South Bend, IN, USA

[3] Mathematics Department, Indian Institute of Technology Delhi, Delhi, India