INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/fld.4009

OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows

Yidong Xia[1,*,†], Jialin Lou[1], Hong Luo[1], Jack Edwards[1] and Frank Mueller[2]

[1]*Department of Mechanical and Aerospace Engineering, North Carolina State University, Raleigh, NC 27695, USA*
[2]*Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA*

SUMMARY

An OpenACC directive-based graphics processing unit (GPU) parallel scheme is presented for solving the compressible Navier–Stokes equations on 3D hybrid unstructured grids with a third-order reconstructed discontinuous Galerkin method. The developed scheme requires the minimum code intrusion and algorithm alteration for upgrading a legacy solver with the GPU computing capability at very little extra effort in programming, which leads to a unified and portable code development strategy. A face coloring algorithm is adopted to eliminate the memory contention because of the threading of internal and boundary face integrals. A number of flow problems are presented to verify the implementation of the developed scheme. Timing measurements were obtained by running the resulting GPU code on one Nvidia Tesla K20c GPU card (Nvidia Corporation, Santa Clara, CA, USA) and compared with those obtained by running the equivalent Message Passing Interface (MPI) parallel CPU code on a compute node (consisting of two AMD Opteron 6128 eight-core CPUs (Advanced Micro Devices, Inc., Sunnyvale, CA, USA)). Speedup factors of up to 24× and 1.6× for the GPU code were achieved with respect to one and 16 CPU cores, respectively. The numerical results indicate that this OpenACC-based parallel scheme is an effective and extensible approach to port unstructured high-order CFD solvers to GPU computing. Copyright © 2015 John Wiley & Sons, Ltd.

Received 10 June 2014; Revised 16 December 2014; Accepted 3 January 2015

KEY WORDS: GPU computing; OpenACC; CUDA; discontinuous Galerkin; compressible flow; Navier–Stokes equations

1. INTRODUCTION

The application of general-purpose graphics processing unit (GPGPU) [1] technology to the CFD solvers has been popular in recent years [2–13]. GPGPU offers an exciting opportunity to significantly accelerate the CFD solvers by offloading compute-intensive portions of the application to the GPU, while the remainder of the computer program still runs on the CPU. From a user's perspective, the solvers simply run much faster.

Among the vendors of GPGPU hardware and software, Nvidia has been an exceptional pioneer in promoting and leading the development of GPGPU technology for the past decade. To the best of the authors' knowledge, numerical methods in CFD solvers that have been attentively studied based on Nvidia's CUDA (Nvidia Corporation) technology include the finite difference methods, spectral difference methods, finite volume methods (FVMs), discontinuous Galerkin methods (DGMs), Lattice Boltzmann method, and more. For example, Elsen *et al.* [14] reported a 3D high-order finite

*Correspondence to: Yidong Xia, currently at Environment Science and Technology Directorate, Idaho National Laboratory, Idaho Falls, ID 83415, USA.
†E-mail: yidong.xia@inl.gov

difference method solver for large calculation on multiblock structured grids, Klöckner *et al.* [15] developed a 3D unstructured high-order nodal DGM solver for the Maxwell's equations, Corrigan *et al.* [16] proposed a 3D FVM solver for compressible inviscid flows on unstructured tetrahedral grids, and Zimmerman *et al.* [17] presented a spectral difference method solver for the Navier–Stokes equations on unstructured hexahedral grids. Nevertheless, the development of CUDA capabilities extended from an existing CFD solver is not a trivial job, because people have to define an explicit layout of the threads on the GPU (numbers of blocks and numbers of threads) for each kernel function [18]. Such a project often requires tremendous hours in programming, as developers have to rewrite all the core content of the source code. Moreover, for a production-level solver, people also need to address both the short-term and long-term investment concerns like the cost and profit, as well as platform portability. These factors can often set people back from investing on GPU computing for their well-established solution products. Even a research-oriented CFD solver is concerned, people may be more inclined to maintain compatibility of their codes across multiple platforms, instead of pursuing performance on one particular platform at the price of being unable to run their codes on other mainstream platforms. Therefore, the development strategy of a CFD solver based on one unique model like CUDA might be a risky long-term investment with unclear prospect of the vendor's own plan. Fortunately, Nvidia is not the sole player in this area. Two other models include OpenCL [19], the currently dominant open GPGPU programming model (but dropped from further discussion because it does not support the FORTRAN programming language) and OpenACC [20], a new programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI.

The OpenACC standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems as well as to closely resemble the OpenMP standard: people simply need to annotate their code to identify the areas that should be accelerated by wrapping with the OpenACC directives and some runtime library routines, without the huge effort to change the original algorithms as to accommodate the code to a specific GPU architecture and compiler. With the OpenACC directives, people benefit not only from easy implementation but also from the freedom to compile the very same code and conduct computations on either CPU or GPU from different vendors. However, compared with CUDA in terms of many desired technical features, the OpenACC standard still lags behind because of vendors' distribution plan (note that Nvidia is among the OpenACC's main supporters). Nevertheless, OpenACC is quickly maturing as an attractive, future GPU parallel programming model for developing portable computer codes and offers a promising approach to minimize the investment in legacy CFD solver by presenting an easy migration path to accelerated computing. Support of OpenACC is available in the commercial compilers from PGI, Cray, and CAPS. OpenUH is an Open64-based open source OpenACC compiler, developed by HPCTools group from the University of Houston. In addition, the GCC (GNU Compiler Collection) project team is also working toward supporting OpenACC in the GCC compilers.

The objective of the effort discussed in the present work is to port an unstructured CFD solver based on a third-order hierarchical Weighted Essentially Non-Oscillatory (WENO) reconstructed DGM [21–24] to GPGPU with OpenACC, for the solution of the 3D compressible Navier–Stokes equations on unstructured hybrid grids. By taking advantage of the OpenACC parallel programming model, the presented scheme requires the minimum code intrusion and algorithm alteration to upgrade a legacy CFD solver without much extra time of effort in programming, resulting in a unified portable code for both CPU and GPU platforms. In addition, a coloring algorithm is used to eliminate memory contention because of threading over the edge-based face loops. A number of inviscid and viscous flow problems are presented to verify and assess the performance of the resulting solver running on GPU.

The outline of the rest of this paper is organized as follows. In Section 2, the governing equations are introduced. In Section 3, the reconstructed DGM is reviewed. In Section 4, the keynotes of porting the underlying discontinuous Galerkin (DG) flow solver to GPU based on OpenACC are discussed in detail. In Section 5, a series of inviscid and viscous flow test cases are presented. Finally, the concluding remarks are given in Section 6.

DOI: 10.1002/fld

2. GOVERNING EQUATIONS

The Navier–Stokes equations governing the unsteady compressible viscous flows can be expressed as

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_k(\mathbf{U})}{\partial x_k} = \frac{\partial \mathbf{G}_k(\mathbf{U}, \nabla \mathbf{U})}{\partial x_k} \tag{1}$$

where the summation convention has been used. The conservative variable vector $\mathbf{U}$, advective flux vector $\mathbf{F}$, and viscous flux vector $\mathbf{G}$ are defined by

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u_i \\ \rho e \end{pmatrix} \qquad \mathbf{F}_j = \begin{pmatrix} \rho u_j \\ \rho u_i u_j + p\delta_{ij} \\ u_j(\rho e + p) \end{pmatrix} \qquad \mathbf{G}_j = \begin{pmatrix} 0 \\ \tau_{ij} \\ u_l \tau_{lj} + q_j \end{pmatrix} \tag{2}$$

Here, ρ, p, and e denote the density, pressure, and specific total energy of the fluid, respectively, and u_i is the velocity of the flow in the coordinate direction x_i. The pressure can be computed from the equation of state

$$p = (\gamma - 1)\rho \left(e - \frac{1}{2} u_i u_i \right) \tag{3}$$

which is valid for perfect gas. The ratio of the specific heats γ is assumed to be constant and equal to 1.4. The viscous stress tensor τ_{ij} and heat flux vector q_j are given by

$$\tau_{ij} = \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij} \qquad q_j = \frac{1}{\gamma - 1} \frac{\mu}{Pr} \frac{\partial T}{\partial x_j} \tag{4}$$

In the aforementioned equations, T is the temperature of the fluid, and Pr is the laminar Prandtl number, which is taken as 0.7 for air. The term μ represents the molecular viscosity, which can be determined through Sutherland's law

$$\frac{\mu}{\mu_0} = \left(\frac{T}{T_0} \right)^{\frac{3}{2}} \frac{T_0 + S}{T + S} \tag{5}$$

where μ_0 is the viscosity at the reference temperature T_0 and $S = 110K$. In addition, the Euler equations can be obtained if the effect of viscosity and thermal conduction are neglected in Equation (1).

3. RECONSTRUCTED DISCONTINUOUS GALERKIN METHOD

Equation (1) can be discretized using a DG finite element formulation [23], which we assume that the readers are familiar with. The HLLC (Harten-Lax-van Leer-Contact) inviscid flux scheme [25] and Bassi–Rebay II viscous flux scheme [26] are used in the present DG method. The numerical polynomial solutions are represented using a Taylor series expansion at the cell centroid and normalized in order to improve the conditioning of the system matrix [27]. For example, the linear polynomial P1 solutions of the underlying DG (P1) method used in the present work consist of the cell-averaged values $\tilde{\mathbf{U}}$ and their normalized first derivatives $\mathbf{U}_x = \left.\frac{\partial \mathbf{U}}{\partial x}\right|_c \Delta x$, $\mathbf{U}_y = \left.\frac{\partial \mathbf{U}}{\partial y}\right|_c \Delta y$, $\mathbf{U}_z = \left.\frac{\partial \mathbf{U}}{\partial z}\right|_c \Delta z$ at the center of the cell:

$$\mathbf{U}_h^{\mathrm{P1}} = \tilde{\mathbf{U}} B_1 + \mathbf{U}_x B_2 + \mathbf{U}_y B_3 + \mathbf{U}_z B_4 \tag{6}$$

where the four basis functions are as follows:

$$B_1 = 1 \qquad B_2 = \frac{x - x_c}{\Delta x} \qquad B_3 = \frac{y - y_c}{\Delta y} \qquad B_4 = \frac{z - z_c}{\Delta z} \tag{7}$$

where $\Delta x = 0.5(x_{\max} - x_{\min})$, $\Delta y = 0.5(y_{\max} - y_{\min})$, and $\Delta z = 0.5(z_{\max} - z_{\min})$. The terms $x_{\max}$, $y_{\max}$, $z_{\max}$ and $x_{\min}$, $y_{\min}$, $z_{\min}$ are the maximum and minimum vertex coordinates of the cell Ω_e,

DOI: 10.1002/fld

respectively. This formulation has a number of attractive, distinct, and useful features. Firstly, the cell-averaged variables and their derivatives are handily available in this formulation. This makes the implementation of both in-cell and inter-cell reconstruction schemes straightforward and simple [28–32]. Secondly, the Taylor basis is hierarchic, which greatly facilitates the implementation of p-multigrid methods [33, 34] and p-refinement. Thirdly, the same basis functions are used for any shapes of elements: tetrahedron, pyramid, prism, and hexahedron. This makes the implementation of DG methods on arbitrary grids straightforward.

By taking advantage of the Taylor basis, a third-order hierarchical WENO reconstruction scheme is recently developed to improve the performance of the underlying second-order DG (P1) method [22]. The procedures can be briefly described in five steps: (i) a quadratic polynomial P2 solution is first reconstructed using a least-squares approach from the underlying linear DG (P1) solution; (ii) this intermediate P2 solution is then used to evaluate the viscous fluxes; (iii) the final second derivatives are obtained through a WENO reconstruction P2, which are necessary to ensure the linear stability of the least-squares reconstructed P2 solution for computing the Euler equations on 3D unstructured tetrahedral grids [21]; (iv) the first derivatives of the quadratic polynomial solution are then reconstructed through a WENO reconstruction at P1 in order to eliminate the spurious oscillations in the vicinity of shocks or discontinuities, thus ensuring the nonlinear stability of the reconstructed DG method; and (v) the final P2 solution is used to evaluate the inviscid fluxes.

Employing the aforementioned hierarchical WENO reconstruction, a system of ODEs in time can be written in a semidiscrete form as

$$\boldsymbol{M}^{\mathrm{P1}} \frac{d\mathbf{U}^{\mathrm{P1}}}{dt} = \mathbf{R}^{\mathrm{P1}}(\mathbf{U}^{\mathrm{P2}}) \tag{8}$$

where $\boldsymbol{M}$ is the mass matrix, $\mathbf{R}$ is the residual vector, and the unknowns to be solved in resulting system of ODEs are still P1 polynomials. We denote this reconstructed DG scheme as rDG (P1P2) in the rest of this paper.

4. OPENACC IMPLEMENTATION

The computation-intensive portion of the rDG (P1P2) method is a time marching loop that repeatedly computes the time derivatives of the conservative variable vector as shown in Equation (8). In the present work, the conservative variable vector $\mathbf{U}$ (solution array) is updated using the multistage total variation diminishing Runge–Kutta explicit time stepping scheme [35, 36] (denoted as TVDRK) in each time step. To activate the computing on GPU, all the required arrays need to be initially allocated on the CPU memory and then copied to the GPU memory before the computation enters time marching. In fact, the data copy between the CPU and its attached GPU needs to be minimized, as it is usually considered to be one of the major overheads in GPU computing. Therefore, in the present code, the data copy of arrays is neither necessary nor allowed within the time marching loop, except for the solution array that can be optionally copied back to the CPU memory every `Ndump` time steps and written to hard disk for the restart and animation purposes. The workflow chart of time stepping is outlined in Figure 1, in which the <ACC> tag denotes an OpenACC acceleration-enabled region and the <MPI> tag means that message passing interface (MPI) routine calls will be invoked in the case that multiple CPUs are used. Compared with the standard DG method, two extra MPI routine calls are required for the rDG (P1P2) method in parallel mode, because of the fact that the solution vector at the partition buffer elements also needs to be updated after each reconstruction call.

The most expensive workload for computing the time derivatives of solutions includes these two procedures: (i) the hierarchical WENO reconstruction that consists of the least-squares quadratic reconstruction (involving both the element and the face loops) and the WENO curvature and gradient reconstructions (involving only the element loops) and (ii) accumulation of the residual vector that consists of internal/boundary integral over the faces and volumetric integral over the elements. In order to achieve a competent speedup, the OpenACC parallel construct directives need to be properly inserted in the code for the compiler to generate the acceleration kernels. In fact, the way to implement OpenACC is very similar to that of OpenMP. The example shown in Figure 2 demon-

DOI: 10.1002/fld

```
! loop over time steps
DO itime = 1, Nstep
<ACC> Predict time-step size
! loop over TVDRK stages
DO istage = 1, Nstage
! P1P2 least-squares reconstruction
<ACC> IF(nreco > 0)&
      CALL reconstruction_ls(...)
! data exchange for partition ghost cells
<MPI> IF(nprcs > 1)&
      CALL exchange(...)
! r.h.s. residual from diffusion
<ACC> IF(nvisc > 0)&
      CALL getrhs_diffusion(...)
```

```
! WENO reconstruction
<ACC> IF(nreco > 0)&
      CALL reconstruction_weno(...)
! data exchange for partition ghost cells
<MPI> IF(nprcs > 1)&
      CALL exchange(...)
! r.h.s. residual from convection
<ACC> CALL getrhs_convection(...)
! update solution vector
<ACC> CALL tvdrk(...)
! data exchange for partition ghost cells
<MPI> IF(nprcs > 1)&
      CALL exchange(...)
ENDDO
ENDDO
```

Figure 1. Workflow for the main loop over the explicit time iterations. WENO, weighted essentially nonoscillatory; ACC, accelerator; MPI, message passing interface; TVDRK, total variation diminishing Runge–Kutta.

```
! OpenMP for CPUs:
! loop over the elements
!$omp parallel
!$omp do
do ie = 1, Nelem
! loop over the Gauss quadrature points
do ig = 1, Ngpel
! contribution to this element
rhsel(*,*,ie) = rhsel(*,*,ie) + flux
enddo
enddo
!$omp end parallel
```

```
! OpenACC for GPUs:
! loop over the elements
!$acc parallel
!$acc loop
do ie = 1, Nelem
! loop over the Gauss quadrature points
do ig = 1, Ngpel
! contribution to this element
rhsel(*,*,ie) = rhsel(*,*,ie) + flux
enddo
enddo
!$acc end parallel
```

Figure 2. An example of loop over the elements. GPU, graphics processing unit.

strates the parallelization of a loop over the elements for collecting contribution to the residual vector `rhsel(1:Ndegr,1:Netot,1:Nelem)`, where `Ndegr` is the number of degree of the approximation polynomial (equal to 1 for P0, 3 for P1, and 6 for P2 in 2D and equal to 1 for P0, 4 for P1, and 10 for P2 in 3D), `Netot` is the number of governing equations of the perfect gas (equal to 4 in 2D and 5 in 3D), `Nelem` is the number of elements, and `Ngpel` is the number of quadrature points over an element. For example, `Ngpel` is equal to 4 in DG (P1), 5 in rDG (P1P2), and 7 in DG (P2) for a tetrahedral element and equal to 8 for DG (P1) and rDG (P1P2) and 27 in DG (P2) for a hexahedral element. Both the OpenMP and OpenACC parallel construct directives can be applied to a readily vectorizable loop like the one in Figure 2, without the need to modify the original code structure. However, because of the unstructured grid topology, the attempt to directly wrap a loop over the dual edges for collecting contribution to the residual vector with either the OpenMP or the OpenACC directives can lead to the so-called ‘race condition’, that is, multiple writes to the same elemental residual vector, and thus result in incorrect values. Unlike in the structured CFD

DOI: 10.1002/fld

solvers [37, 38], this kind of 'race condition' issue is typically associated to the threading of dual-edge loops in unstructured CFD solvers for both the node-centered and the cell-centered schemes, which is not strange at all to those who have the experience of developing parallel unstructured CFD solvers based on OpenMP. In the present study, a summary of two 'contention-free' vectorization strategies for threading the dual-edge loops is described and assessed as follows:

4.1. Element-based algorithm

In this scheme, the internal/boundary face integral is incorporated into a grand loop over the elements as proposed by Corrigan *et al.* [16] for the FVMs. So, as a result, all the workload-intensive computations are wrapped in element-wise loops, which are perfect for threading. However, a significant overhead associated to this algorithm is its redundant computation for the dual edges. According to [16], the performance of the developed finite volume solver based on CUDA was only advantageous for computation of single precision and became much worse in the case of double precision. Unfortunately, this algorithm does not meet our design goals mainly for two reasons. Firstly, the DG methods require an inner loop over the quadrature points `Ngpfa` (equal to 3 in DG (P1), 4 in rDG (P1P2), and 7 in DG (P2) for a triangular face and equal to 4 in DG (P1) and rDG (P1P2) and 9 in DG (P2) for a quadrilateral face) for computing the face integrals in dual-edge computation. For example, it accounts for at least 50% of the gross computing time as in the second-order DG (P1) method. Note that `Ngpfa` could be a larger number in the case of higher-order DG methods, which could lead to a much more severe overhead if the workload of such computation is doubled. Secondly, the implementation of this algorithm indicates a major rework in the code structure, which would not only require tremendous hours in programming but also completely ruin the performance of the equivalent CPU code. Note that most of the modern unstructured CFD solvers adopt an edge-based algorithm in dual-edge computations.

4.2. Edge-based algorithm

In the edge-based algorithm, a coloring scheme consisting of face renumbering and grouping can be used to eliminate the 'race condition'. The advantage of using this scheme is that it does not require any change to the original code structures. The coloring scheme is designed to divide all of the faces into a number of groups by ensuring that any two faces that belong to a common element never fall into the same group, so that the face loop in each group can be threaded free of 'race condition'. Figure 3 shows an example where an extra do-construct that trips over those groups sequentially is nested on top of the original internal face loop. Therefore, the inner do-construct that trips over the internal faces can be threaded without 'race condition'. In fact, this type of algorithm is widely used for threaded computing in unstructured CFD solvers. Details of the implementation can be found in an abundance of literature, for example, [39]. According to our study, the number of groups for a grid is usually between 6 and 8 according to a wide range of test cases, indicating only a few minor overheads in repeatedly launching and terminating the OpenACC acceleration kernels for the loop over the face groups. This kind of overheads is typically associated to GPU computing but not the case for threaded computing on CPU. Nevertheless, a remarkable feature in this design approach lies in the fact that it allows the legacy CPU code to be recovered when the OpenACC directives are dismissed in the preprocessing stage of compilation. Therefore, the use of this edge-based coloring algorithm has resulted in a unified source code for both the CPU and GPU computing.

To sum up from the foregoing discussion, the edge-based algorithm is considered to suit well in the present work for its simplicity, as it can be quickly implemented without any major change in the legacy code. Overall, it is applied in the internal/boundary face integrals when computing the residuals, as well as in the other procedures that contain the loop over faces like the least-squares quadratic reconstruction and the prediction of allowable local time-step size for each element.

5. NUMERICAL EXAMPLES

The source code is written in Fortran 90 (International Business Machines Corporation) and compiled by the PGI Accelerator (The Portland Group, Inc., Lake Oswego, OR, USA) with OpenACC +

DOI: 10.1002/fld

```
! OpenMP for CPUs (no race condition):

! loop over the groups
Nfac1 = Njfac
do ipass = 1, Npass_ift
Nfac0 = Nfac1 + 1
Nfac1 = fpass_ift(ipass)
! loop over the edges
!$omp parallel
!$omp do
do ifa = Nfac0, Nfac1
! left  element
iel = intfac(1,ifa)
! right element
ier = intfac(2,ifa)
! loop over Gauss quadrature points
do ig = 1, Ngpfa
! contribution to the left element
rhsel(*,*,iel) = rhsel(*,*,iel) - flux
! contribution to the right element
rhsel(*,*,ier) = rhsel(*,*,ier) + flux
enddo
enddo
!$omp end parallel
enddo
```

```
! OpenACC for GPUs (no race condition):

! loop over the groups
Nfac1 = Njfac
do ipass = 1, Npass_ift
Nfac0 = Nfac1 + 1
Nfac1 = fpass_ift(ipass)
! loop over the edges
!$acc parallel
!$acc do
do ifa = Nfac0, Nfac1
! left  element
iel = intfac(1,ifa)
! right element
ier = intfac(2,ifa)
! loop over Gauss quadrature points
do ig = 1, Ngpfa
! contribution to the left element
rhsel(*,*,iel) = rhsel(*,*,iel) - flux
! contribution to the right element
rhsel(*,*,ier) = rhsel(*,*,ier) + flux
enddo
enddo
!$acc end parallel
enddo
```

Figure 3. An example of loop over the edges. GPU, graphics processing unit.

OpenMPI development suite. An Nvidia Tesla K20c GPU card (Nvidia Corporation) (2496 stream processors, 5-GB memory, and 200 GB/s bandwidth) is used to verify and assess the performance of the resulting GPU code. This GPU card is attached to a CPU compute node, consisting of two AMD Opteron 6128 eight-core processors (Advanced Micro Devices, Inc.). The minimum compilation flags required for generating the double-precision, optimized, and Nvidia's GPU-supported executables are as follows:

$$-\mathrm{r8} - \mathrm{O3} - \mathrm{acc} - \mathrm{ta} = \mathrm{nvidia,time,cc20}$$

To evaluate the speedup, we compare the unit time $\mathrm{T_{unit}}$ measured by running the GPU code on the K20c with that measured by running the equivalent CPU code on the compute node. $\mathrm{T_{unit}}$ is defined as

$$\mathrm{T_{unit}} = \frac{\mathrm{T_{run}}}{\mathtt{Nstage} \times \mathtt{Nstep} \times \mathtt{Nelem}} \times 10^6 \quad \text{(microseconds)} \tag{9}$$

where $\mathrm{T_{run}}$ refers to the wall clock time measured only for completing the entire time marching loop with a given number of time steps `Nstep`, excluding the start-up procedures, initial/end data translation, and solution file dumping. Note that `Nstage` $=$ `3` because of the use of three-stage total variation diminishing Runge–Kutta3 for time stepping. In addition, the well-known TauBench

DOI: 10.1002/fld

was run with one process (-np), 250,000 DOFs per process (-n), and 10 pseudosteps (-s)

$$\text{mpirun} - \text{np1./TauBench} - \text{n250000} - \text{s10}$$

three times to obtain an average wall clock time $T_{tau} = 36$s, along with 0.439580 GFLOPS. The work unit is then defined as T_{run}/T_{tau}, which is a widely accepted performance indicator for unstructured grids on CPU [40]. Note that there has not been a commonly accepted GPU benchmark testing for unstructured CFD solvers. Therefore, although speedup factors based on specific hardware models are not particularly preferable in a strict sense, yet it is not uncommon that they are used in literature, for example, [10].

5.1. Inviscid subsonic flow past a sphere

In this test case, an inviscid subsonic flow past a sphere at a free stream Mach number of $M_\infty = 0.5$ is considered. Computation is conducted on a sequence of three successively refined tetrahedral grids as displayed in Figure 4(a)–4(c). The cell size is halved between two consecutive grids. Note that only a quarter of the configuration is modeled because of the symmetry of the problem. The

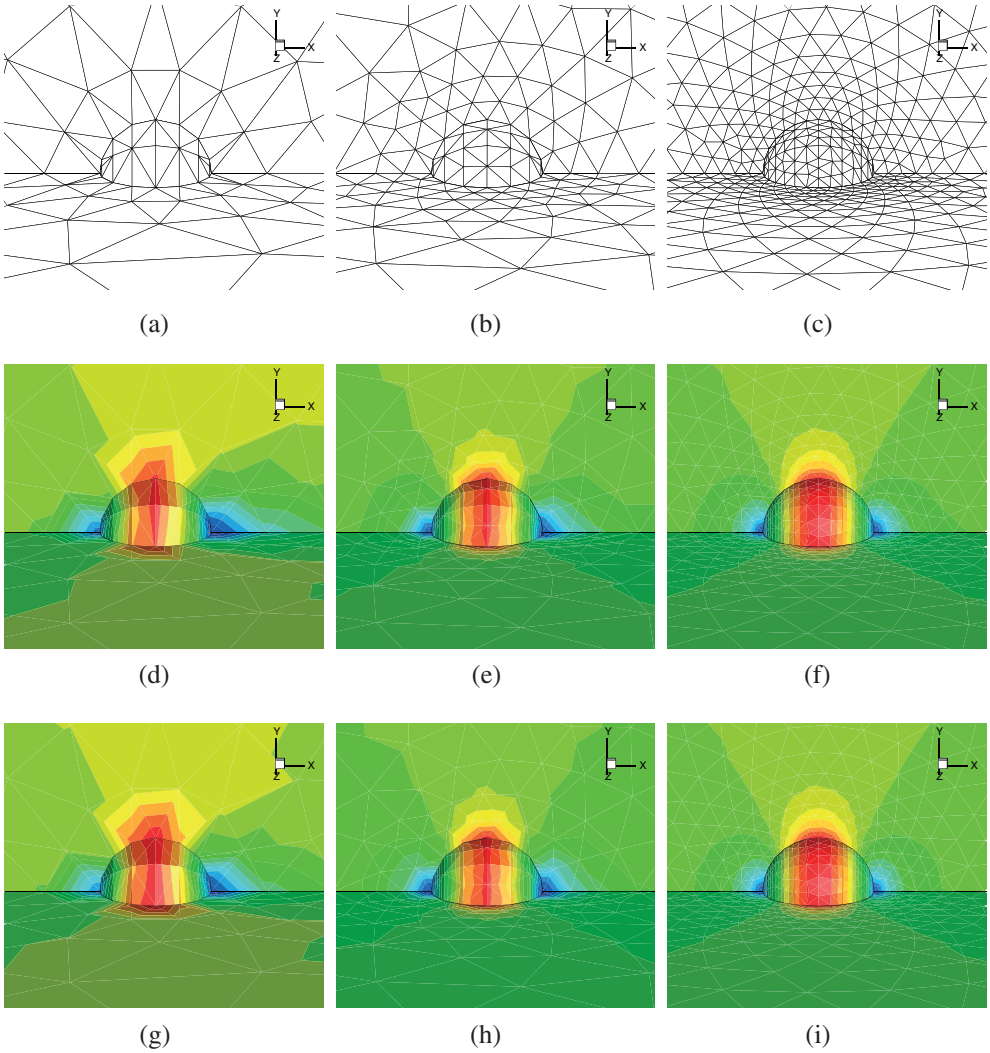


Figure 4. Subsonic flow past a sphere at $M_\infty = 0.5$: (a)–(c) the three successively refined tetrahedral grids used in the verification test, (d)–(f) computed Mach number contours obtained by DG (P1), and (g)–(i) computed Mach number contours obtained by rDG (P1P2).

DOI: 10.1002/fld

computation is started with a uniform flow field and terminated at a sufficiently large total number of time steps to obtain a steady-state solution, as shown in Figure 4(d)–4(i). The following L^2 norm of the entropy production is used as the error measurement for the steady-state inviscid flow problems:

$$\|\varepsilon\|_{L^2(\Omega)} = \sqrt{\int_\Omega \varepsilon^2 \, d\Omega} = \sqrt{\sum_{i=1}^{Nelem} \int_{\Omega_i} \varepsilon^2 \, d\Omega} \tag{10}$$

where the entropy production ε is defined as

$$\varepsilon = \frac{S - S_\infty}{S_\infty} = \frac{p}{p_\infty}\left(\frac{\rho_\infty}{\rho}\right)^\gamma - 1 \tag{11}$$

Note that the entropy production, where the entropy is defined as $S = (p/\rho)^\gamma$, is a very good criterion to measure the accuracy of the numerical solutions, because the flow under consideration is isentropic. The discretization errors are presented in Table I. As one can see, DG (P1) and rDG (P1P2) both achieved a formal order of accuracy of convergence, being 2.00 and 3.01, respectively, convincingly demonstrating the benefits of using the rDG method over its underlying baseline DG method. In addition, a handmade `diff` program with a user-defined absolute error tolerance of 1.0×10^{-12} indicates that the GPU code and the CPU code produced the identical solution on each grid. Next, a strong scaling test is carried out for rDG (P1P2) on a sequence of four successively refined tetrahedral grids, as shown in Table II. The total number of time steps is set to be 100 for all of these four grids. As one has observed, the OpenACC GPU code does not gain advantage over the 16 CPU processors for a small-scale problem like 2426 elements. With adequate grid size like 124,706 and 966,497 elements, the advantage of GPU is then fully demonstrated, as speedup factors of up to 22.6× and 1.49× have been achieved by comparing with the CPU code running on one and 16 CPU processors, respectively. Finally, variations of the TauBench work unit versus DOFs are shown in Figure 5, providing a relatively subjective indicator to compare with for any other explicit third-order DG solvers.

5.2. Viscous subsonic flow past a sphere

In this test case, we consider a viscous subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$ and a low Reynolds number of $Re_\infty = 118$ based on the diameter of the sphere. Firstly, computation is conducted on a coarse grid consisting of 119,390 tetrahedral elements as shown in Figure 6(a), in order to verify the implementation of OpenACC parallel scheme. Note that only half

Table I. Discretization errors and convergence rates obtained on the three successively refined tetrahedral grids for inviscid subsonic flow past a sphere at $M_\infty = 0.5$.

`Nelem`	L^2-norm (P1)	$\mathcal{O}(h^2)$	L^2-norm (P1P2)	$\mathcal{O}(h^3)$
535	−0.1732E+01	—	−0.196E+01	—
2426	−0.2302E+01	1.90	−0.284E+01	2.92
16,467	−0.2933E+01	2.09	−0.377E+01	3.09

Table II. Timing measurements obtained by reconstructed discontinuous Galerkin (P1P2) for subsonic flow past a sphere at $M_\infty = 0.5$.

	T_{unit} (microseconds)			Speedup	
`Nelem`	One GPU	One CPU	16 CPUs	Versus one CPU	Versus 16 CPUs
2426	6.7	58.9	4.9	8.8×	0.73×
16,467	3.6	60.9	4.2	17.0×	1.18×
124,706	3.1	60.9	4.3	19.6×	1.40×
966,497	2.9	66.3	4.3	22.6×	1.49×

GPU, graphics processing unit.

DOI: 10.1002/fld

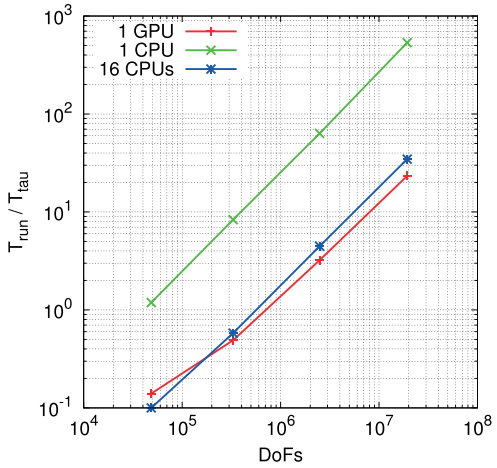
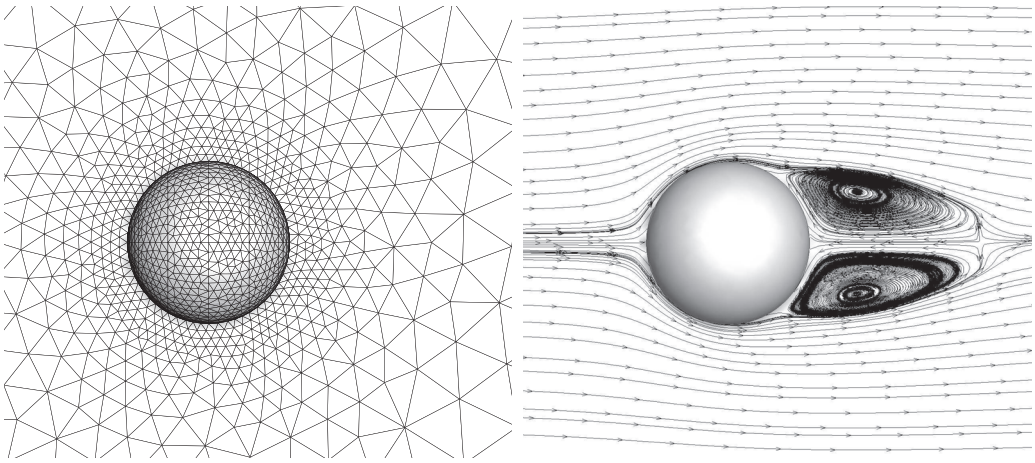


Figure 5. Plot of the TauBench work unit versus DOFs (20 in each tetrahedron for reconstructed discontinuous Galerkin (P1P2)) for computing inviscid subsonic flow past a sphere at $M_\infty = 0.5$. GPU, graphics processing unit.



(a) (b)

Figure 6. Viscous flow past a sphere at $M_\infty = 0.5$ and $Re_\infty = 118$: (a) the tetrahedral grid in the verification test and (b) the computed streamtraces on the symmetry plane.

of the configuration is modeled because of the symmetry of the problem. The no-slip, adiabatic boundary conditions are prescribed to the solid wall. Figure 6(b) displays the computed steady-state velocity streamtraces on the symmetry plane obtained by rDG (P1P2). As one can observe, the two trailing vortices are visually identical and symmetric to the center line. A `diff` check with an absolute error tolerance of 1.0×10^{-12} indicates that the GPU code and the CPU code produced the identical solution data. Secondly, a strong scaling test is conducted with the timing measurements presented in Table III. Speedup factors of up to 18.5 are obtained w.r.t. one CPU processor and 1.48 w.r.t. the 16 processors. It is interesting to find that the speedup factors obtained for the Navier–Stokes equations are lower than those for the Euler equations in the previous case, although the computational intensity in this case is obviously higher. In fact, the major latency is due to the code structure that the viscous and inviscid flux calculations are divided into two separate procedures, because the WENO reconstructed quadratic polynomials are only needed for computing the inviscid residuals, as shown in Figure 1. Therefore, the overheads in acceleration kernels within the r.h.s. computation are doubled. Indeed, the code could render higher efficiency if we chose to cluster the least-squares reconstruction and WENO reconstruction at the head of the r.h.s. process and

DOI: 10.1002/fld

Table III. Timing measurements obtained by reconstructed discontinuous Galerkin (P1P2) for a viscous subsonic flow past a sphere at $M_\infty = 0.5$ and $Re_\infty = 118$.

	T_{unit} (microseconds)			Speedup	
`Nelem`	One GPU	One CPU	16 CPUs	Versus one CPU	Versus 16 CPUs
200,416	4.9	86.6	6.8	17.8×	1.41×
925,925	4.6	85.7	6.9	18.5×	1.48×

GPU, graphics processing unit.

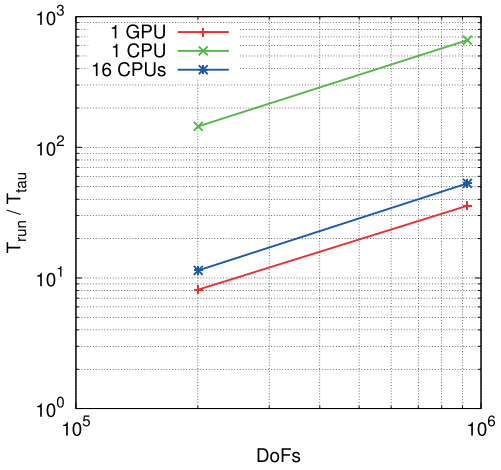


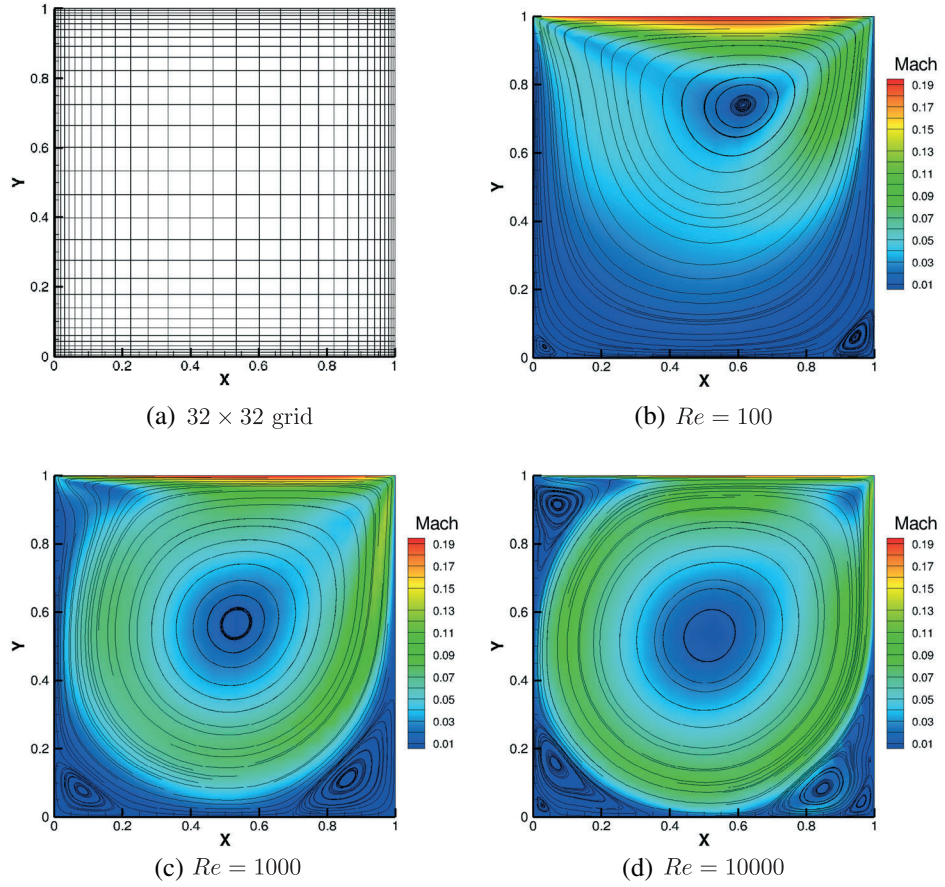
Figure 7. Plot of the TauBench work unit versus DOFs (20 in each tetrahedron reconstructed discontinuous Galerkin (P1P2)) for a viscous subsonic flow past a sphere at $M_\infty = 0.5$ and $Re_\infty = 118$. GPU, graphics processing unit.

merge the viscous and inviscid flux calculations into one face integral and one volumetric integral. However, study shows that the solution accuracy would be affected if we did so. Finally, variations of the TauBench work unit versus DOFs obtained by running 100 three-stage time steps are shown in Figure 7, which can be compared with for any other explicit third-order DG solvers for computing the 3D Navier–Stokes equations on tetrahedral grids.

5.3. Quasi-2D lid-driven square cavity

A quasi-2D lid-driven square cavity laminar flow at a series of Reynolds numbers of $Re = 100$, 1000, and 10,000 is considered in this test case. The cavity dimensions are 1 unit in the x and y directions and 0.1 unit in the z direction. Computation is first conducted on a coarse hexahedral grid, which consists of $32\times32\times2$ grid points as shown in Figure 8(a), in an attempt to (i) verify the implementation of OpenACC for hexahedral elements and (ii) demonstrate the advantage of rDG (P1P2), as the classical reference data by Ghia *et al.* [41] can be used to assess the accuracy of the computed velocity profiles. The grid points are clustered near the walls in the x and y directions, and the grid spacing is geometrically stretched away from the wall with the minimum value $h_{\min} = 0.005$ (equivalent to $y^+ = 3.535$). On the bottom and side walls, the no-slip, adiabatic boundary conditions are prescribed. Along the top 'lid', the no-slip, adiabatic boundary conditions along with a lid velocity $\mathbf{v}_b = (0.2, 0, 0)$ are prescribed. The computed steady-state velocity streamtraces obtained by rDG (P1P2) are displayed in Figure 8(b)–8(d), demonstrating its ability to accurately resolve all the major vortices on this coarse grid. Figures 9–11 display the profiles of the normalized velocity components u/u_B and v/u_B obtained by DG(P1) and rDG (P1P2) that are plotted along the y and x center lines, respectively. The profiles by a second-order compressible FVM based on a WENO reconstruction [42], namely rDG (P0P1), is also presented. Overall, rDG (P1P2) has demonstrated superior accuracy over the other two methods provided with such sparse grid resolution, especially in the case of high Reynolds numbers. A `diff` check with an absolute error tolerance of 1.0×10^{-12}

DOI: 10.1002/fld



(a) 32×32 grid (b) $Re = 100$

(c) $Re = 1000$ (d) $Re = 10000$

Figure 8. A quasi-2D lid-driven square cavity flow at $\mathbf{v}_B = (0.2, 0, 0)$ and a series of Re: (a) 32×32 grid, (b) $Re = 100$, (c) $Re = 1000$, and (d) $Re = 10,000$

.

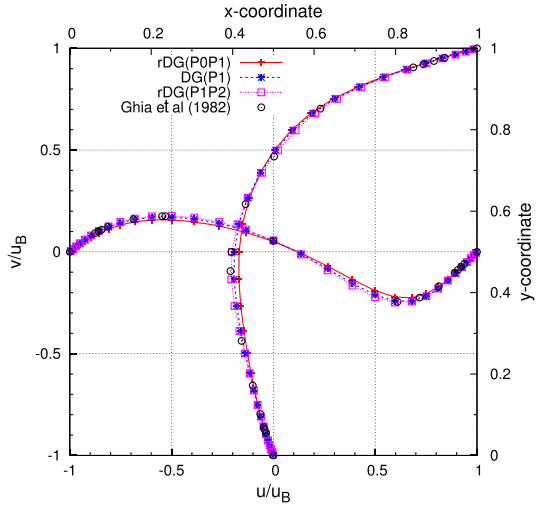


Figure 9. Profiles of the normalized velocity components u/u_B and v/u_B on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid-driven square cavity at $\mathbf{v}_B = (0.2, 0, 0)$ and $Re = 100$. rDG, reconstructed discontinuous Galerkin; DG, discontinuous Galerkin.

DOI: 10.1002/fld

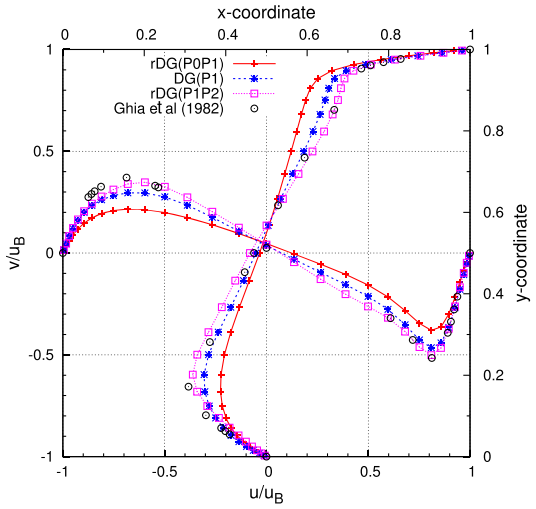


Figure 10. Profiles of the normalized velocity components u/u_B and v/u_B on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid-driven square cavity at $\mathbf{v}_B = (0.2, 0, 0)$ and $Re = 1000$. rDG, reconstructed discontinuous Galerkin; DG, discontinuous Galerkin.

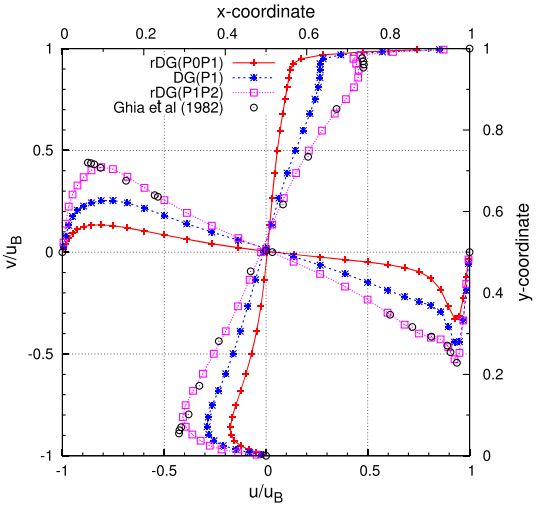


Figure 11. Profiles of the normalized velocity components u/u_B and v/u_B on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid-driven square cavity at $\mathbf{v}_B = (0.2, 0, 0)$ and $Re = 10,000$. rDG, reconstructed discontinuous Galerkin; DG, discontinuous Galerkin.

indicates that the GPU code and the CPU code produced the identical solution data. Secondly, a strong scaling test is designed and conducted by running 100 three-stage time steps on two hexahedral grids, which contain 500,000 and 1,000,000 elements, respectively. The timing measurements are presented in Table IV. Speedup factors of up to 19.0 and 1.50 were achieved by comparing the unit running time obtained on the K20c GPU with those by the one and 16 CPU processors. In addition, variations of the TauBench work unit versus DOFs are presented in Figure 12. Overall, we can see that the developed OpenACC GPU code renders consistent performance on different types of elements.

5.4. Transonic flow over a Boeing 747 aircraft

In the final test case, a transonic flow past a complete Boeing 747 aircraft (Boeing Commercial Airplanes, Renton, WA, USA) at a free-stream Mach number of $M_\infty = 0.85$ and an angle of attack of

DOI: 10.1002/fld

Table IV. Timing measurements obtained by reconstructed discontinuous Galerkin (P1P2) for a quasi-2D lid-driven square cavity at $\mathbf{v}_B = (0.2, 0, 0)$ and $Re = 10,000$.

	T_{unit} (microseconds)			Speedup	
`Nelem`	One GPU	One CPU	16 CPUs	Versus one CPU	Versus 16 CPUs
500,000	5.9	109.7	8.7	18.7×	1.48×
1,000,000	5.8	109.6	8.7	19.0×	1.50×

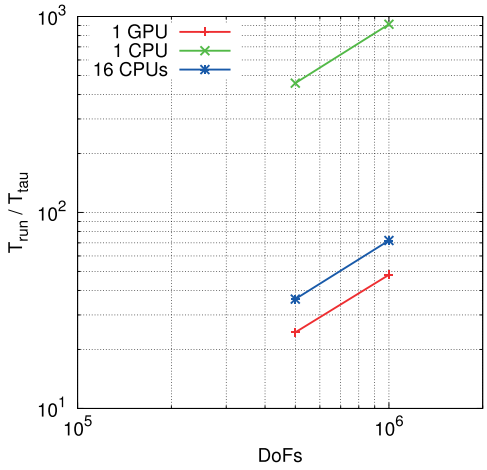
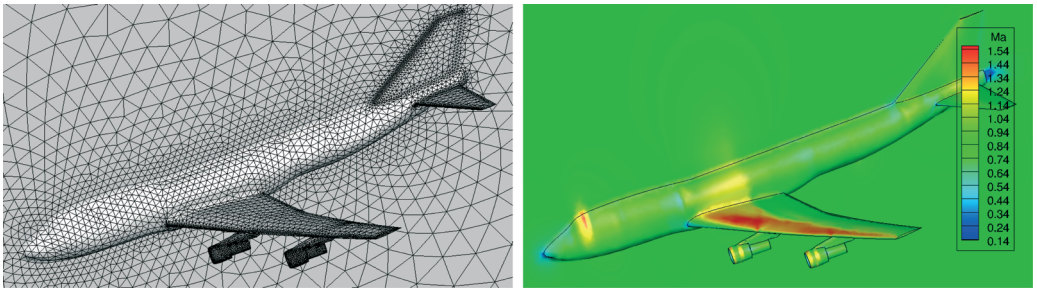


Figure 12. Plot of the TauBench work unit versus DOFs (20 in each hexahedron for reconstructed discontinuous Galerkin (P1P2) with Taylor basis) for a quasi-2D lid-driven square cavity at $\mathbf{v}_B = (0.2, 0, 0)$ and $Re = 10,000$. GPU, graphics processing unit.



(a) (b)

Figure 13. Transonic flow over a Boeing 747 aircraft at $M_\infty = 0.85$ and $\alpha = 2°$: (a) surface unstructured triangular meshes and (b) Mach number contours obtained by reconstructed discontinuous Galerkin (P1P2). GPU, graphics processing unit.

$\alpha = 2°$ is presented in order to assess the performance of the OpenACC GPU code for computing complex geometries. The Boeing 747 configuration includes the fuselage, wing, horizontal and vertical tails, under'wing pylons, and flow-through engine nacelle. Computation is first conducted on a tetrahedral grid containing 253,577 elements, as shown in Figure 13(a). Note that only the half-span airplane is modeled because of the symmetry of the problem. The computed steady-state Mach number contours obtained by rDG (P1P2) are illustrated in Figure 13(b). One can observe that the shock waves on the upper surface of the wing are well captured, confirming the accuracy, robustness, and efficiency of our method for computing complicated flows of practical importance. A `diff` check with the absolute error tolerance of 1.0×10^{-12} indicates that the GPU code produced the identical solution data to those by the equivalent CPU code. Secondly, a scaling test for the K20c GPU card is conducted by running 100 three-stage time steps, with the timing measurements obtained by rDG

DOI: 10.1002/fld

Table V. Timing measurements obtained by reconstructed discontinuous Galerkin (P1P2) for inviscid transonic flow over a Boeing 747 aircraft at $M_\infty = 0.85$ and $\alpha = 2^\circ$.

	T_{unit} (microseconds)			Speedup	
Nelem	One GPU	One CPU	16 CPUs	Versus one CPU	Versus 16 CPUs
253,577	3.8	81.3	5.5	21.2×	1.43×
1,025,170	3.5	83.1	5.5	24.5×	1.57×

GPU, graphics processing unit.

(P1P2) presented in Table V. Speedup factors of up to 24.5× and 1.57× have been achieved for the GPU code by comparing with the CPU code running on one and full 16 processors of the CPU compute node, respectively. Above all, the highest speedup factors observed in this test case are similar to those in the first test case, indicating the consistent and stable performance of the resulting OpenACC GPU code for computing various flow conditions and geometric configurations.

6. CONCLUSION AND OUTLOOK

In this study, an OpenACC directive-based parallel scheme has been presented for the GPU computing of an unstructured CFD solver based on a third-order WENO reconstructed DGM. Indeed, compared with the more mature and dominating techniques like CUDA, the current OpenACC specification and compilers have not yet been well defined and optimized although active development and improvement are underway. Therefore, it is not surprising that a fine-tuned CUDA code could usually outperform the equivalent OpenACC code as of today. Nevertheless, as we have stressed, the biggest benefits by adopting OpenACC for our CFD solvers are still evident: it requires the minimum code intrusion and algorithm alteration to upgrade a legacy unstructured CFD solver with the GPU computing capability without much extra effort in programming, thus could save tremendous work hours in code development and maintenance. Numerical experiments on a number of flow problems have been conducted to verify the implementation of the developed scheme. The results of timing measurements indicate that this OpenACC-based parallel scheme is able to significantly accelerate the solving for the equivalent legacy CPU code. A following paper is being prepared with a focus on the development, implementation, and assessment of multi-GPU parallelism for the reconstructed DGM.

ACKNOWLEDGEMENTS

The authors would also like to acknowledge the support for this work provided by the Basic Research Initiative program of The Air Force Office of Scientific Research. Dr. F. Fariba and Dr. D. Smith served as the technical monitors.

REFERENCES

1. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, Vol. 26, Wiley Online Library, 2007; 80–113.
2. Brandvik T, Pullan G. Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 2007; **221**(12):1745–1748.
3. Brandvik T, Pullan G. Acceleration of a 3d Euler solver using commodity graphics hardware. *46th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, USA, 2008; 2008–607.
4. Goddeke D, Buijssen SH, Wobker H, Turek S. GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. *International Conference on High Performance Computing & Simulation, 2009. HPCS'09*, IEEE, Leipzig, Germany, 2009; 12–21.
5. Cohen J, Molemaker MJ. A fast double precision CFD code using CUDA. *21st International Conference on Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, Moffett Field, California, USA, 2009; 414–429.

DOI: 10.1002/fld

6. Phillips EH, Zhang Y, Davis RL, Owens JD. Rapid aerodynamic performance prediction on a cluster of graphics processing units. *Proceedings of the 47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition*, 2009; 2009–565.
7. Thibault JC, Senocak I. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. *Proceedings of the 47th AIAA Aerospace Sciences Meeting including The Horizons Forum and Aerospace Exposition*, Orlando, Florida, USA, 2009; 2009–758.
8. Jacobsen DA, Thibault JC, Senocak I. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. *48th AIAA Aerospace Sciences Meeting and Exhibit including The New Horizon Forum and Aerospace and Exposition*, Vol. 16, 2010–522.
9. Michéa D, Komatitsch D. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International* 2010; **182**(1):389–402.
10. Corrigan A, Camelli F, Löhner R, Mut F. Porting of an edge-based CFD solver to GPUs. *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, Orlando, Florida, USA, 2010–523.
11. Jespersen DC. Acceleration of a CFD code with a GPU. *Scientific Programming* 2010; **18**(3):193–201.
12. Asouti VG, Trompoukis XS, Kampolis IC, Giannakoglou KC. Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *International Journal for Numerical Methods in Fluids* 2011; **67**(2):232–246.
13. Corrigan A, Camelli F, Löhner R, Mut F. Semi-automatic porting of a large-scale fortran CFD code to GPUs. *International Journal for Numerical Methods in Fluids* 2012; **69**(2):314–331.
14. Elsen E, LeGresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics* 2008; **227**(24):10148–10161.
15. Klöckner A, Warburton T, Bridge J, Hesthaven JS. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics* 2009; **228**(21):7863–7882.
16. Corrigan A, Camelli F, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids* 2011; **66**(2):221–229.
17. Zimmerman B, Wang Z, Visbal M. High-order spectral difference: verification and acceleration using GPU computing. *21st AIAA Computational Fluid Dynamics Conference*, San Diego, California, USA, 2013; 2013–2491.
18. Jin H, Kellogg M, Mehrotra P. Using compiler directives for accelerating CFD applications on GPUs. In *OpenMP in a Heterogeneous World*. Springer: Berlin, Germany, 2012; 154–168.
19. Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering* 2010; **12**(3):66.
20. Wienke S, Springer P, Terboven C, Mey D. OpenACC first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*. Springer: Berlin, Germany, 2012; 859–870.
21. Luo H, Xia Y, Li S, Nourgaliev R. A Hermite WENO reconstruction-based discontinuous Galerkin method for the Euler equations on tetrahedral grids. *Journal of Computational Physics* 2012; **231**(16):5489–5503.
22. Luo H, Xia Y, Spiegel S, Nourgaliev R, Jiang Z. A reconstructed discontinuous Galerkin method based on a hierarchical WENO reconstruction for compressible flows on tetrahedral grids. *Journal of Computational Physics* 2013; **236**:477–492.
23. Xia Y, Luo H, Frisbey M, Nourgaliev R. A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids. *Computers & Fluids* 2014; **98**:134–151.
24. Xia Y, Luo H, Nourgaliev R. An implicit Hermite WENO reconstruction-based discontinuous Galerkin method on tetrahedral grids. *Computers & Fluids* 2014; **96**:406–421.
25. Batten P, Leschziner MA, Goldberg UC. Average-state Jacobians and implicit methods for compressible viscous and turbulent flows. *Journal of Computational Physics* 1997; **137**(1):38–78.
26. Bassi F, Rebay S. Discontinuous Galerkin solution of the Reynolds-averaged Navier-Stokes and κ-ω turbulence model equations. *Computers & Fluids* 2005; **34**(4–5):507–540.
27. Luo H, Baum JD, Löhner R. A discontinuous Galerkin method using Taylor basis for compressible flows on arbitrary grids. *Journal of Computational Physics* 2008; **227**(20):8875–8893.
28. Luo H, Luo L, Nourgaliev R, Mousseau V, Dinh N. A reconstructed discontinuous Galerkin method for the compressible Navier-Stokes equations on arbitrary grids. *Journal of Computational Physics* 2010; **229**(19): 6961–6978.
29. Luo H, Luo L, Ali A, Nourgaliev R, Cai C. A parallel, reconstructed discontinuous Galerkin method for the compressible flows on arbitrary grids. *Communication in Computational Physics* 2011; **9**(2):363–389.
30. Luo H, Luo L, Nourgaliev R. A reconstructed discontinuous Galerkin method for the Euler equations on arbitrary grids. *Communications in Computational Physics* 2012; **12**(5):1495–1519.
31. Zhang LP, Liu W, He LX, Deng XG, Zhang HX. A class of hybrid DG/FV methods for conservation laws II: two dimensional cases. *Journal of Computational Physics* 2012; **231**(4):1104–1120.
32. Luo H, Xia Y, Nourgaliev R. A class of reconstructed discontinuous Galerkin methods in computational fluid dynamics. *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C2011)*, Brazil, 2011; 1–17.
33. Luo H, Baum JD, Löhner R. A fast, p-multigrid discontinuous Galerkin method for compressible flows at all speeds. *AIAA Paper* 2006; **110**:2006.
34. Luo H, Baum JD, Löhner R. A p-multigrid discontinuous Galerkin method for the Euler equations on unstructured grids. *Journal of Computational Physics* 2006; **211**(2):767–783.

DOI: 10.1002/fld

35. Cockburn B, Hou S, Shu CW. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws IV: the multidimensional case. *Journal of Mathematical Physics* 1990; **55**:545–581.
36. Cockburn B, Shu CW. The Runge-Kutta discontinuous Galerkin method for conservation laws V: multidimensional system. *Journal of Computational Physics* 1998; **141**:199–224.
37. Pickering BP, Jackson CW, Scogland TR, Feng W, Roy CJ. Directive-based GPU programming for computational fluid dynamics. *52nd AIAA Aerospace Sciences Meeting AIAA Paper*, National Harbor, Maryland, USA, 2014; 2014–1131.
38. Luo L, Edwards JR, Luo H, Mueller F. Performance assessment of a multi-block incompressible Navier-Stokes solver using directive-based GPU programming in a cluster environment. *52nd AIAA Aerospace Sciences Meeting AIAA Paper*, National Harbor,Maryland, USA, 2014; 2014–1130.
39. Löhner R. *Applied Computational Fluid Dynamics Techniques: An Introduction Based on Finite Element Methods*. John Wiley & Sons: Hoboken, New Jersey, USA, 2008.
40. Wang Z, Fidkowski K, Abgrall R, Bassi F, Caraeni D, Cary A, Deconinck H, Hartmann R, Hillewaert K, Huynh H. High-order CFD methods: current status and perspective. *International Journal for Numerical Methods in Fluids* 2013; **72**(8):811–845.
41. Ghia U, Ghia KN, Shin CT. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics* 1982; **48**(3):387–411.
42. Xia Y, Liu X, Luo H. A finite volume method based on a WENO reconstruction for compressible flows on hybrid grids. *52nd AIAA Aerospace Sciences Meeting AIAA Paper*, National Harbor, Maryland, USA, 2014; 2014–0939.

DOI: 10.1002/fld