# On the Multi-GPU Computing of a Reconstructed Discontinuous Galerkin Method for Compressible Flows on 3D Hybrid Grids

**CONFERENCE PAPER** · JUNE 2014

DOI: 10.2514/6.2014-3081



**6 AUTHORS**, INCLUDING:

Yidong Xia

Idaho National Laboratory

**31** PUBLICATIONS **75** CITATIONS

SEE PROFILE

Lixiang Luo

North Carolina State University

**23** PUBLICATIONS **61** CITATIONS

SEE PROFILE

Hong Luo

Sichuan Normal University

**97** PUBLICATIONS **1,382** CITATIONS

SEE PROFILE

# On the Multi-GPU Computing of a Reconstructed Discontinuous Galerkin Method for Compressible Flows on 3D Hybrid Grids

Yidong Xia *, Jialin Lou †, Lixiang Luo ‡, Hong Luo § and Jack Edwards ¶‡

*Department of Mechanical and Aerospace Engineering*

*North Carolina State University, Raleigh, NC 27695-7910, United States*

and

Frank Mueller ‖

*Department of Computer Science*

*North Carolina State University, Raleigh, NC 27695-8206, United States*

**A multi-GPU accelerated, third-order, reconstructed discontinuous Galerkin method, namely RDG(P1P2), has been developed based on the OpenACC directives for compressible flows on 3D hybrid grids. The present scheme requires minimum intrusion and algorithm alteration to an existing CPU code, which renders an efficient design approach for upgrading a legacy CFD solver with the GPU-computing capability while maintaining its portability across multiple platforms. The grid partitioning is performed according to the number of GPUs, and loaded equally on each GPU. Communication between the GPUs is achieved via the host-based MPI. A face renumbering and grouping algorithm is used to eliminate memory contention due to vectorized computing over the face loops on each individual GPU. A series of inviscid and viscous flow problems have been presented for the verification and scaling test, demonstrating excellent scalability of the resulting GPU code. The numerical results indicate that this parallel RDG(P1P2) method is a cost-effective, high-order DG method for scalable computing on GPU clusters.**

## I. Introduction

Nowadays computational fluid dynamics (CFD) has been one of the most important applications that run on supercomputers. However the gap between the capabilities of the traditional high performance computing technology based on CPU (central processing unit) and the complexity of the simulation problems to be solved continues to widen. Therefore a faster and more cost-effective hardware model as well as a new programming model has been sought in order to meet the needs of high performance computing in future. Fortunately the GPGPU[22] technology, which stands for general-purpose graphics processing unit, offers an exciting opportunity to significantly accelerate the high performance computing not only in CFD applications, but also in many other computational sciences, and thus is one of the major growing computation models in recent years. In principle, the GPGPU offers unprecedented application performance 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, applications simply run much faster.

Among the major vendors of GPGPU hardware and software, Nvidia has been an exceptional pioneer in promoting and leading the development of GPGPU technology in the past decade. In particular, the Nvidia

*Postdoctoral Research Associate, AIAA Member. Corresponding Author: yxia2@ncsu.edu

†Graduate Student

‡Postdoctoral Research Associate.

§Professor, AIAA Associate Fellow.

¶Professor, AIAA Associate Fellow.

‖Professor

CUDA application programming interface (API) and CUDA-enabled graphical accelerators are recognized as the most popular parallel programming model and platform in computational sciences nowadays. As a matter of fact, most recent research efforts[4,5,14,8,23,26,15,21,9,16,1,10] in the investigation and development of GPU-accelerated CFD solvers have been conducted based on the CUDA programming model and Nvidia hardware. To the best of the authors' knowledge, the numerical methods that have been attentively studied for GPU parallel computing include the finite difference methods (FDMs), spectral difference methods (SDMs), finite volume methods (FVMs), discontinuous Galerkin methods (DGMs), Lattice Boltzmann method (LBMs), and more. For example, Elsen *et al.*[12] reported a 3D high-order FDM solver for large calculation on multi-block structured grids; Klöckner *et al.*[18] developed a 3D unstructured high-order nodal DGM solver for the Maxwell's equations; Corrigan *et al.*[11] proposed a 3D FVM solver for compressible inviscid flows on unstructured tetrahedral grids; Zimmerman *et al.*[36] presented an SDM solver for the Navier-Stokes equations on unstructured hexahedral grids.

In general, the development of a CUDA-based CFD solver can be categorized in two types: 1) a brand new code design, and 2) extension from an existing code. In the first type people simply start from scratch, which is more often the case of fundamental study of a numerical model on GPU computing. In the second type, however, upgrading a legacy CFD code of either production level or research level by using the Nvidia CUDA model is not likely a trivial job, since people have to define an explicit layout of the threads on the GPU (numbers of blocks, numbers of threads) for each kernel function.[17] Such a design project indicates a huge human resource that has to be spent rewriting all the core content of the source code. As in the development of a production-level code, people also need to address both the short and long term investment concerns like the cost and benefit, and more importantly, the platform portability. These concerns can often set people back from utilizing GPU capabilities for their solution products. Even at the research level, people may be more inclined to maintain multi-platform compatibility for their codes, instead of pursuing performance improvement on one particular platform at the price of losing the code portability on other platforms. Therefore the development of a CFD code based on the CUDA model might spell a 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:[25] the currently dominant open GPGPU programming model (but dropped from further discussion since it does not support Fortran); and OpenACC:[28] a new open parallel programming standard based on directives.

OpenACC is designed to closely resemble OpenMP: 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. By using OpenACC for programming, people benefit not only from easy implementation of the directives but also the freedom to compile the very same code and conduct computations on either CPU or GPU from different vendors. However on the other side, OpenACC still lags behind CUDA in terms of many desired advanced features due to vendors' distribution plan (note that Nvidia is among the OpenACC's main supporters). Nevertheless OpenACC becomes available as an attractive parallel programming model for the development of a portable and unified code, and offers a promising approach to minimize the investment in legacy CFD codes by presenting an easy migration path to accelerated computing.

The objective of the effort discussed in the present work is to develop a multi-GPU accelerated, third-order, reconstructed discontinuous Galerkin method, namely RDG(P1P2), for the solution of the compressible Navier-Stokes equations on 3D hybrid grids. This work is based on a class of reconstruction-based RDG(P$n$P$m$) methods,[19,20,29,34] which are recently developed in order to improve the overall performance of the underlying standard DG(P$n$) methods without significant extra costs in terms of computing time and storage requirement. Part of the solution modules in a well verified and validated RDG flow solver have already been upgraded with the capability of single-GPU computing based on the OpenACC directives in our prior work,[32] The present design requires minimum intrusion and algorithm alteration to an existing CPU code, which renders an efficient approach to upgrading a legacy CFD solver with the GPU-computing capability while maintaining its portability across multiple platforms. The grid partitioning is performed according to the number of GPUs, and loaded equally on each GPU. Communication between the GPUs is achieved via the host-based MPI. A face renumbering and grouping algorithm is used to eliminate memory contention due to vectorized computing over the face loops on each individual GPU. A series of inviscid and viscous flow problems have been presented for the verification and scaling test, demonstrating excellent scalability of the resulting GPU code. The numerical results indicate that this parallel RDG(P1P2) method is a cost-effective, high-order DG method for scalable computing on GPU clusters.

The outline of the rest of this paper is organized as follows. In Section II, the governing equations are briefly introduced. In Section III, the discontinuous Galerkin spatial discretization is described. In Section IV, the keynotes of porting a discontinuous Galerkin flow solver to GPUs based on the OpenACC directives is discussed in detail. In Section V, a series of inviscid and viscous flow test cases are presented. Finally the concluding remarks are given in Section VI.

## II. 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 $\rho$, $p$, 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 $\gamma$ is assumed to be constant and equal to 1.4. The viscous stress tensor $\tau_{ij}$ 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 above equations, $T$ is the temperature of the fluid, $Pr$ the laminar Prandtl number, which is taken as 0.7 for air. $\mu$ represents the molecular viscosity, which can be determined through Sutherlands law

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

where $\mu_0$ 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 Eq. 1.

## III. Discontinuous Galerkin Spatial Discretization

The governing equations in Eq. 1 can be discretized using a discontinuous Galerkin finite element formulation. We assume that the domain $\Omega$ is subdivided into a collection of non-overlapping arbitrary elements $\Omega_e$ in 3D, and then introduce the following broken Sobolev space $V_h^p$

$$V_h^p = \left\{ v_h \in \left[ L^2(\Omega) \right]^m : v_h|_{\Omega_e} \in \left[ V_p^m \right] \forall \Omega_e \in \Omega \right\} \tag{6}$$

which consists of discontinuous vector polynomial functions of degree $p$, and where $m$ is the dimension of the unknown vector and $V_p$ is the space of all polynomials of degree $\leq p$. To formulate the discontinuous Galerkin method, we introduce the following weak formulation, which is obtained by multiplying Eq. 1 by a test function $\mathbf{W}_h$, integrating over an element $\Omega_e$, and then performing an integration by parts: find $\mathbf{U}_h \in V_h^p$ such as

$$\frac{d}{dt} \int_{\Omega_e} \mathbf{U}_h \mathbf{W}_h d\Omega + \int_{\Gamma_e} \mathbf{F}_k \mathbf{n}_k \mathbf{W}_h d\Gamma - \int_{\Omega_e} \mathbf{F}_k \frac{\partial \mathbf{W}_h}{\partial x_k} d\Omega = \int_{\Gamma_e} \mathbf{G}_k \mathbf{n}_k \mathbf{W}_h d\Gamma - \int_{\Omega_e} \mathbf{G}_k \frac{\partial \mathbf{W}_h}{\partial x_k} d\Omega, \quad \forall \mathbf{W}_h \in V_h^p \tag{7}$$

where $\mathbf{U}_h$ and $\mathbf{W}_h$ are represented by piecewise polynomial functions of degrees $p$, which are discontinuous between the cell interfaces, and $\mathbf{n}_k$ the unit outward normal vector to the $\Gamma_e$: the boundary of $\Omega_e$. Assume that $B_i$ is the basis of polynomial function of degrees $p$, this is then equivalent to the following system of $N$ equations,

$$\frac{d}{dt}\int_{\Omega_e}\mathbf{U}_h B_i d\Omega + \int_{\Gamma_e}\mathbf{F}_k\mathbf{n}_k B_i d\Gamma - \int_{\Omega_e}\mathbf{F}_k\frac{\partial B_i}{\partial x_k}d\Omega = \int_{\Gamma_e}\mathbf{G}_k\mathbf{n}_k B_i\ d\Gamma - \int_{\Omega_e}\mathbf{G}_k\frac{\partial B_i}{\partial x_k}d\Omega,\quad 1\le i\le N \tag{8}$$

where $N$ is the dimension of the polynomial space. Since the numerical solution $\mathbf{U}_h$ is discontinuous between element interfaces, the interface fluxes are not uniquely defined. This scheme is called discontinuous Galerkin method of degree $p$, or in short notation DG($p$) method. By simply increasing the degree $p$ of the polynomials, the DG methods of corresponding higher order are obtained. In the present work, the HLLC scheme[3] and Bassi-Rebay II scheme[2] are used for evaluating the inviscid and viscous fluxes, respectively.

By moving the second and third terms to the right-hand-side (r.h.s.) in Eq. 8, we will arrive at a system of ordinary differential equations (ODEs) in time, which can be written in semi-discrete form as

$$\boldsymbol{M}\frac{d\mathbf{U}}{dt} = \mathbf{R}(\mathbf{U}) \tag{9}$$

where $\boldsymbol{M}$ is the mass matrix and $\mathbf{R}$ is the residual vector. The present work employs a third-order, WENO reconstructed scheme to improve the overall performance of the underlying second-order DG(P1) method,[19] without much extra cost in computing time and memory requirement.

## IV. OpenACC Implementation

### A. Overview

The computation-intensive portion of this reconstructed discontinuous Galerkin method is a time marching loop which repeatedly computes the time derivatives of the conservative variable vector as shown in Eq.9. In the present work, the conservative variable vector is updated using the explicit, three-stage, TVD Runge-Kutta time stepping scheme[6,7] (TVDRK3) within each time iteration. To enable GPU computing, all the required arrays are first allocated on the CPU memory and initialized before the computation enters the main loop. These arrays are then copied to the GPU memory, most of which will not need to be copied back the CPU memory. In fact, the data copy between the CPUs and GPUs, usually considered to be one of the major overheads in GPU computing, needs to be minimized in order to improve the efficiency. The workflow of time iterations is outlined in Table 1, in which the <ACC> tag denotes an OpenACC acceleration-enabled region, and the <MPI> tag means that MPI routine calls are needed in the case that multiple CPUs or GPUs are invoked. Compared with the standard DG method, two extra MPI routine calls are required for the RDG method in parallel mode, due to the fact that the solution vector at the partition ghost elements need to be updated after each reconstruction scheme.

### B. Multiple-GPU communication

Nowadays, a top-of-the-line GPGPU card like the Nvidia Tesla K series can contain over two thousand stream processors, yet its memory size is relatively small in comparison with a regular CPU compute node. For example, a Tesla K20c Accelerator has 2496 stream processors and 5GB in memory — sufficient for scientific computing and business modeling in various disciplines. However, such a memory size is still far from enough even for some basic CFD simulation problems. In order to extend the memory, we can choose a more recent product like the Tesla K40 Accelerator (2880 stream processors, 12GB memory). But the pricing of such a new device is still prohibitive ($4440 before tax), which would make GPU computing a rather disadvantageous alternative to CPU computing in terms of cost-effectiveness. Instead, the use of a cluster of legacy GPGPU cards appears a more economical and practical choice for large-scale CFD computing. Unlike the case of single-GPU computing that requires no communication between the devices within the main loop of time steps, the major difference in designing a multi-GPU parallel model lies in the fact that the inter-GPU communication needs to be taken into account. In the present work, we adopt a parallelism scheme based on a message passing interface (MPI) programming paradigm, where the METIS library is used for the partitioning of a grid into subdomain grids of approximately the same size. Consequently, the data exchange is required based on the MPI library, and the workflow is sketched in Table

Table 1: Workflow for the main loop over the explicit time iterations.

```
!! loop over time steps
DO itime = 1, ntime

  <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
```

2, where the continuous memory corresponding to the solution vector of the partition ghost elements (from (`Nelem`+1)-th element to (`Nadje`)-th element) is first copied from the GPUs to the host CPUs, and then copied back to the GPUs after a non-blocking communication procedure. As one has observed, the overall inter-GPU communication procedure has to go through the host CPUs. In fact, the so-called GPUDirect[27,24] techniques, which are intended to realize direct communication between the GPUs, has yet to mature. In addition, most of the recently reported GPUDirect solutions would more or less require some unique software customization. Furthermore, even if these techniques are achievable within a durable effort on our current HPC resource, the resulting code is unlikely to be portable across the platforms. Therefore those are the main reasons that we do not anticipate the GPUDirect techniques to be incorporated into the framework of our OpenACC-based GPU code, unless more fundamental improvement in terms of hardware structures and OpenACC specifications would take place in the GPGPU industry.

### C. Parallelism for acceleration regions

The most expensive workload when computing the time derivatives of solutions $d\mathbf{U}/dt$ includes the following two categories of procedures:

1. Reconstruction of the second derivatives
   - P1P2 least-squares reconstruction (involves both element and face loops)
   - P2 WENO reconstruction (involves on element loops)
2. Accumulation of the r.h.s. residual vector in Eq. 9
   - Face integral over the dual-edges
   - Domain integral over the elements

Thus these procedures need to be properly ported to acceleration kernels by using the OpenACC parallel construct directives. In fact , the way to add OpenACC directives in a legacy code is very similar to that

Table 2: Workflow for the data exchange and synchronization procedure.

```
!! data exchange for partition ghost elements
SUBROUTINE exchange(...)

  !! copy partition ghost elements from GPU to CPU
  !$acc update host( unkno(:, :, Nelem+1:Nadje) )

  !! initialize receive
  DO i = 1,Ncomm
    s = exprc(i)
    dtype = irecv(i)
    CALL mpi_irecv(...)
  ENDDO

  !! send out partition-adjacent elements
  DO i = 1,Ncomm
    r = exprc(i)
    dtype = isend(i)
    call mpi_isend(...)
  ENDDO

  !! wait for all sends and receives to complete
  CALL mpi_waitall(...)


  !! copy partition ghost elements from CPU to GPU
  !$acc update device( unkno(:, :, Nelem+1:Nadje) )

END SUBROUTINE exchange
```

of OpenMP. The example shown in Table 3 demonstrates 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 (= 1 for P0, 3 for P1 and 6 for P2 in 2D; = 1 for P0, 4 for P1 and 10 for P2 in 3D), `Netot` the number of governing equations of the perfect gas (= 4 in 2D, 5 in 3D), `Nelem` the number of elements, and `Ngp` the number of Gauss quadrature points over an element. Both the OpenMP and OpenACC parallel construct directives can be applied to a readily vectorizable loop like in Table 3 without the need to modify the original code. However due to the unstructured grid topology, the

Table 3: An example of loop over the elements.

```
!! OpenMP for CPUs:

!! loop over the elements
!$omp parallel
!$omp do
do ie = 1, Nelem

  !! loop over the Gauss quadrature points
  do ig = 1, Ngp

    !! 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, Ngp

    !! contribution to this element
    rhsel(*,*,ie) = rhsel(*,*,ie) + flux

  enddo

enddo
!$acc end parallel
```

attempt to directly wrap a loop over the dual-edges for collecting contribution to the residual vector with either the OpenMP or OpenACC directives can lead to the so-called "race condition", that is, multiple writes to the same elemental residual vector, and thus result in the incorrect values. In general, two approaches are available to eliminate the "race condition" issue for vectorized computing on unstructured grids as described below:

**`Approach I`**

One approach is to incorporate the dual-edge computation into the loop over the elements as proposed by Corrigan *et al.*[11] for the finite volume methods. In this approach, all the workload-intensive computations are wrapped in element-wise loops that are perfectly vectorizable, thus no "race condition"

would occur for the resulting code. However, a major overhead associated to this approach is its redundant computation for the dual-edges. According to Reference,[11] the performance of the developed finite volume solver based on CUDA was only advantageous in single-precision GPU computation, and became much worse in double-precision. In fact, this approach would never meet our design goals for two reasons. For the first, the discontinuous Galerkin methods require an inner loop over the Gauss quadrature points `Ngp` for computing the face integrals in dual-edge computation, which account for at least 50% of the gross computing time as in the second-order DG(P1) method. Furthermore, `Ngp` could be a larger number in the case of higher-order DG methods, therefore resulting in a huge overhead if the workload of such computation is doubled. For the second, the implementation of this approach indicates a major change in the code structure, which is not only costly in programming, but also would completely ruin the performance of the equivalent CPU code.

**Approach II**

Alternatively, the "race condition" can be eliminated with a moderate amount of work by adopting a mature algorithm of face renumbering and grouping. This algorithm is designed to divide all the faces into a number of groups by ensuring that any two faces that belong to a common element never fall in the same group, so that the face loop in each group can be vectorized without "race condition". An example is shown in Table 4, where an extra do-construct that loops over these groups is nested on top of the original loop over the internal faces, and executed sequentially. The inner do-construct that loops over the internal faces is vectorized without the "race condition" issue. In fact, this kind of algorithm is widely used for vectorized computing on unstructured grids with OpenMP. The implementation details can be found in an abundance of literature. The number of groups for each subdomain grid is usually between 6 and 8 according to a wide range of test cases, indicating some 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 for the code if parallelized by OpenMP for CPU computing. Nevertheless, the most favorable feature in this design approach is that it allows the original CPU code to be recovered when the OpenACC directives are dismissed in the pre-processing stage of compilation. Therefore, the use of this face renumbering and grouping algorithm will result in a unified source code for both the CPU and GPU computing on unstructured grids.

To sum up from the discussion above, the parallelism in Approach II can suit well in the present work, for its simplicity and portability to quickly adapt into the original source code without any major change in the legacy programming structures. It is applied for the face integrals as well as some other procedures that require the loop over faces like P1P2 least-squares reconstruction and evaluation of the local time-step sizes.

## V. Numerical examples

Performance of the developed GPU code based on OpenACC was measured on the North Carolina State University's research-oriented cluster `ARC`, which has 1728 CPU cores on 108 compute nodes integrated by Advanced HPC. All machines are 2-way SMPs with AMD Opteron 6128 (Magny Core) processors with 8 cores per socket (16 cores per node). The GPGPU cards used in the present study are shown in Figs. 1(a) and 1(b), with their details listed in Table. 5. Note that each GPGPU card is attached to one compute node on the `ARC` cluster. The source code was written in Fortran 90 and compiled with the PGI Accelerator with OpenACC (version 13.9) + OpenMPI (version 1.5.1) development suite. The minimum compilation flags required for generating the double-precision, optimized, Nvidia GPU-accelerated executables are: `-r8 -O3 -acc -ta=nvidia,time,cc20`

To evaluate the speedup of GPU versus CPU, we compared the running time measured by using the GPU code on one Nvidia Tesla K20c card with that measured by using the equivalent CPU code on one compute node (16 CPU cores). To assess the scalability of our code on multiple GPUs, we performed a weak scaling experiment on up to eight Nvidia Tesla C2050 cards for each test case. The unit time $T_{unit}$ is calculated as

$$T_{unit} = \frac{T_{\text{wall-clock}} \times \texttt{Ngpus}}{\texttt{Ntime} \times \texttt{Nelem}} \times 10^6 \quad \text{(microsecond)}$$

where $T_{\text{wall-clock}}$ refers to the wall-clock time recorded for completing the entire time marching loop with a given number of time steps `Ntime` by using an amount of `Ngpus` GPU cards, not including the start-up procedures, initial/end data translation, and solution file dumping.

Table 4: An example of loop over the edges.

```
!! OpenMP for CPUs (without 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, Ngp

      !! 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 (without 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, Ngp

      !! 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
```

Three numerical examples are introduced in the rest of this section. In each of these examples, the following test suite was carried out to verify and validate the developed flow solver:

- A verification test with an absolute error tolerance of $1.0 \times 10^{-12}$ that compares the numerical results obtained by the GPU code with those by the equivalent CPU code.
- A weak scaling test that assesses how the solution time varies with the number of GPU cards for a fixed problem size per GPU card.

In addition, the readers are suggested to refer to the authors' prior work[32] where the results of strong scaling test based on the competition between a single K20c GPU card and a sixteen-core CPU compute node can be found.

Table 5: Part of the Nvidia GPGPU resource on the NCSU's `ARC` cluster (donated by Nvidia)

| Type | Amount | Stream processors | Memory | Bandwidth (GB/sec) |
|---|---|---|---|---|
| Tesla C2050 | 9 | 448 | 3 GB | 144 |
| Tesla K20c | 3 | 2496 | 5 GB | 200 |

### A. Inviscid flow past a sphere

In the first test case, an inviscid subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$ is considered in order to assess the performance of the developed code for solving the Euler equations. First the computation is conducted on four Nvidia Tesla C2050 GPU cards to verify if the GPU code can deliver the identical numerical solutions to those by the equivalent CPU code. A sequence of three successively refined tetrahedral grids, which were used in the authors' prior works[31,33] for the verification and validation purposes, are displayed in Figs. 2(a) – 2(c). The cell size is halved between two consecutive grids. Note that only a quarter of the configuration is modeled due to symmetry of the problem. The computation is started


(a) (b)

Figure 1: The Nvidia Tesla GPGPUs used in the present work: (a) C2050 (448 stream processors, 3GB memory); (b) K20c (2496 stream processors, 5GB memory).

with a uniform flow field, and terminated at a sufficiently large total number of time iterations for each grid to obtain a steady-state solution. Figs. 2(d) – 2(f) and Figs. 2(g) – 2(i) show the computed Mach number contours on the surface meshes obtained by DG(P1) and RDG(P1P2), respectively. The following $L^2$ norm of the entropy production is used as the error measurement for the steady-state inviscid flow problems:

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

where the entropy production $\varepsilon$ is defined as

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

Note that the entropy production, where the entropy is defined as $S = (p/\rho)^\gamma$, is a very good criterion to measure accuracy of the numerical solutions, since the flow under consideration is isentropic. The quantitative measurement of the discretization errors as shown in Table 6. As one has observed, both the DG(P1) and RDG(P1P2) methods 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. Most importantly, a hand-made `diff` program with a defined absolute error tolerance of $1.0 \times 10^{-12}$ indicates that the GPU code and the CPU code produced the identical solution data on each grid.

Table 6: Discretization errors and convergence rates obtained on the three successively refined tetrahedral grids for inviscid subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$.

| Grids | Elements | $L^2$ norm (P1) | Order (P1) | $L^2$ norm (P1P2) | Order (P1P2) |
|---|---|---|---|---|---|
| Coarse | 535 | -0.1732E+01 | – | -0.196E+01 | – |
| Medium | 2,426 | -0.2302E+01 | 1.895 | -0.284E+01 | 2.924 |
| Fine | 16,467 | -0.2933E+01 | 2.094 | -0.377E+01 | 3.094 |

Secondly a weak scaling test is carried out on a sequence of four successively refined tetrahedral grids, which contain approximately half million, one million, two million, and four million elements, respectively, as shown in Table 7. These four grids correspond to the use of one, two, four, and eight Nvidia Tesla C2050 GPU cards respectively, ensuring an approximately fixed problem size per GPU card. The total number of time iterations is set to be $10,000$ for all of these four grids. The detailed timing measurements are presented in Table 7, showing the statistics of unit running time and parallel efficiency obtained on each grid. In addition, Figs. 3(a) and 3(b) display the variations of the unit running time and parallel efficiency with

respect to the number of GPUs, indicating that both the DG(P1) and RDG(P1P2) methods have achieved good parallel efficiency in the case of eight GPUs, being 87.0% and 87.7% respectively. The primary loss of efficiency in multi-GPU mode is due to the overheads in GPU-to-CPU and CPU-to-GPU data copies, and MPI communication and synchronization between the host CPUs. Above all, this parallel RDG solver based on the OpenACC directives exhibits a competitive scalability for computing inviscid flow problems on multiple GPUs.

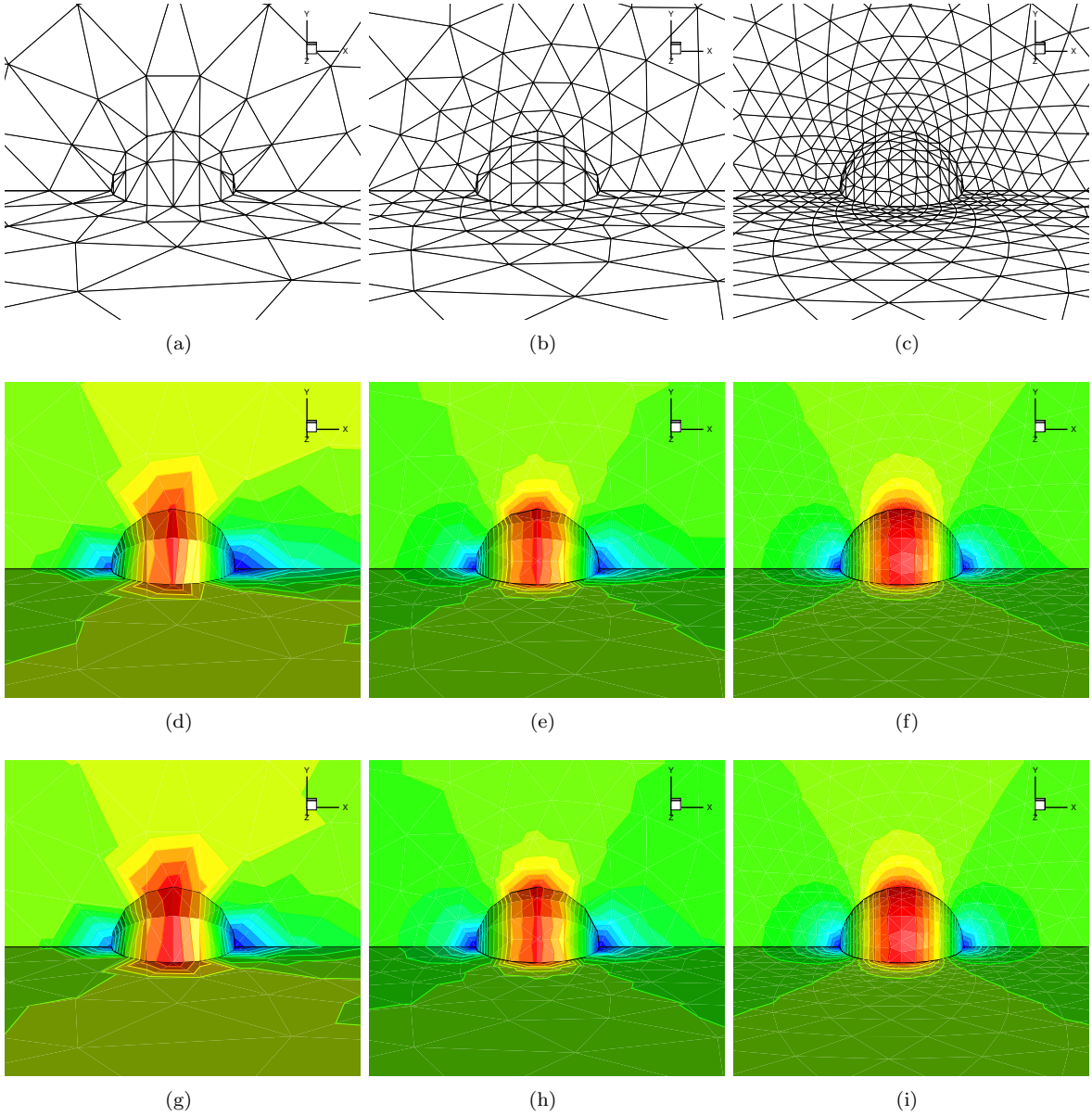


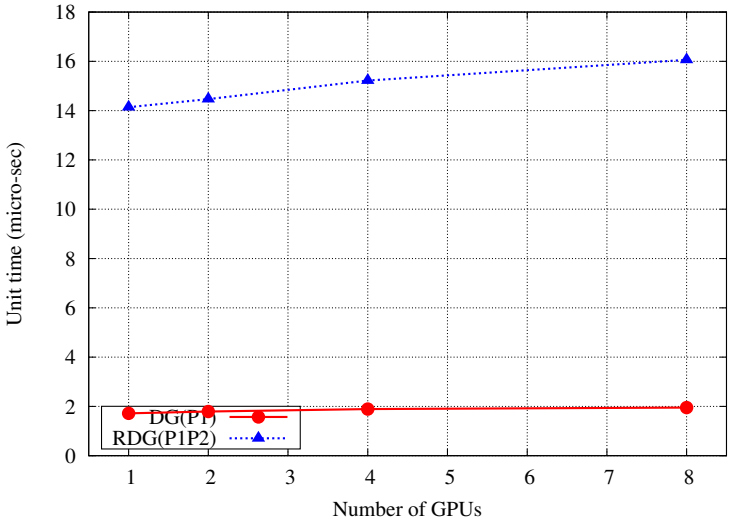

Figure 2: Subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$: (a) – (c) Surface triangular meshes of the three successively refined tetrahedral grids used in the verification test; (d) – (f) Computed Mach number contours obtained by DG(P1) on the surface meshes; (g) – (i) Computed Mach number contours obtained by RDG(P1P2) on the surface meshes.

### B. Viscous flow past a sphere

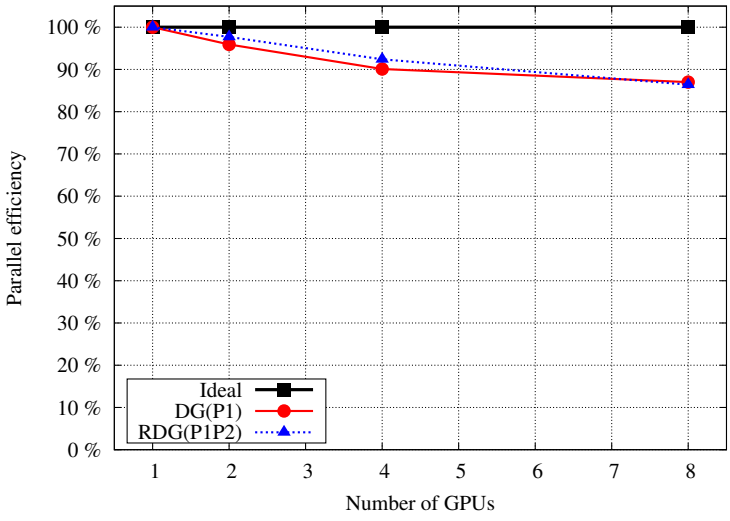
In the second test case, a viscous flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$, and a low Reynolds number of $Re = 118$ based on the diameter of the sphere is considered in order to assess

Table 7: Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for inviscid subsonic flow past a sphere.

| | | | Storage (GB) | | Unit time (ms) | | Parallel efficiency | |
|---|---|---|---|---|---|---|---|---|
| Grids | Elements | GPU's | P1 | P1P2 | P1 | P1P2 | P1 | P1P2 |
| Level 1 | 501,972 | ×1 | 1.4 | 1.8 | 1.72 | 14.14 | – | – |
| Level 2 | 1,015,570 | ×2 | 2.8 | 3.6 | 1.79 | 14.47 | 95.9% | 97.7% |
| Level 3 | 1,999,386 | ×4 | 5.6 | 7.3 | 1.89 | 15.22 | 90.1% | 92.4% |
| Level 4 | 4,029,430 | ×8 | 11.2 | 14.6 | 1.95 | 16.06 | 87.0% | 86.4% |



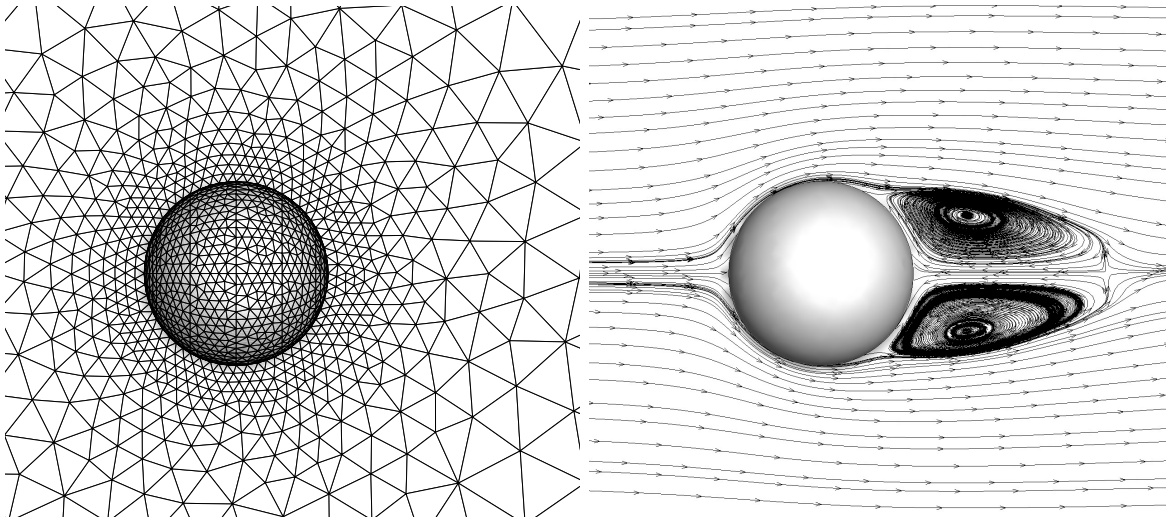

(a)




(b)

Figure 3: Plot of the timing measurements for inviscid subsonic flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.

the performance of the developed code for solving the Navier-Stokes equations. First the computation is conducted on four Nvidia Tesla C2050 GPU cards to verify if the GPU code can deliver the identical numerical solutions to those by the equivalent CPU code. A coarse tetrahedral grid consisting of 119,390 elements, 22,530 grid points and 4,511 boundary grid points is used in this computation, as shown in Fig. 4(a). One can observe the coarseness of the triangular surface meshes near the sphere wall region. Note that only half of the configuration is modeled due to the symmetry of the problem. The computation is started with a uniform free-stream flow field and no-slip boundary conditions on the solid wall, and terminated at a sufficiently large total number of time iterations to obtain a steady-state solution. The computed velocity streamtraces by RDG(P1P2) is plotted on the symmetry plane as shown in Fig. 4(b). As one can observe, the two trailing vortices are visually identical and symmetric to the center-line. In addition, a `diff` check with an absolute error tolerance of $1.0 \times 10^{-12}$ indicates that the GPU code and the CPU code produced the identical solution data in this verification test.



(a) (b)

Figure 4: Viscous flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$ and a Reynolds number of $Re = 118$: (a) The tetrahedral grid in the verification test. (b) The computed streamtraces on the symmetry plane.

Table 8: Timing measurements of strong scaling obtained by RDG(P1P2) on a Nvidia Tesla K20c GPU card for a viscous flow past a sphere.

| | Unit time (ms) | | | Speedup | |
|---|---|---|---|---|---|
| Elements | 1 GPU | 1 CPU | 16 CPUs | vs. 1 CPU | vs. 16 CPUs |
| 200,416 | 14.6 | 259.9 | 20.5 | 17.8 | 1.41 |
| 925,925 | 13.9 | 257.2 | 20.6 | 18.5 | 1.48 |

Secondly a strong scaling test is carried out on one Nvidia Tesla K20c GPU card, which has competed against one and full sixteen cores of a dual AMD Opteron 6128 compute node, respectively. The computations were conducted on a coarse grid consisting of 200,416 elements and a fine grid consisting of 925,995 elements respectively, with the detailed timing measurements presented in Table 8. A speedup factor of up to 18.5 is obtained with respect to one CPU core, and 1.48 with respect to the full sixteen CPU cores. It is interesting to discern that the speedup factors obtained for solving the Navier-Stokes equations are lower than those for the Euler equations,[32] although the computational intensity is obviously higher in the former case. This is mainly due to the overheads in initializing doubled acceleration kernels each time the r.h.s. residual vector is calculated, since the contribution from viscous and inviscid flux calculations is implemented in two separate

edge loops in the current RDG solver.

Thirdly a weak scaling test is carried out on a sequence of four successively refined tetrahedral grids, which contain approximately half million, one million, two million, and four million elements, respectively. These four grids correspond to the use of one, two, four, and eight Nvidia Tesla C2050 GPU cards respectively, ensuring an approximately fixed problem size per GPU card. The total number of time iterations is set to be $10,000$ for all of these four grids. with the detailed timing measurements presented in Table 9. Figs. 5(a) and 5(b) display the variations of the unit running time and parallel efficiency with respect to the number of GPUs respectively. The parallel efficiency ratios of 90.3% and 91.2% were obtained for DG(P1) and RDG(P1P2) in the case of eight GPUs, respectively, demonstrating the good scalability of the developed RDG solver on multiple GPUs for computing viscous flow problems on tetrahedral grids.

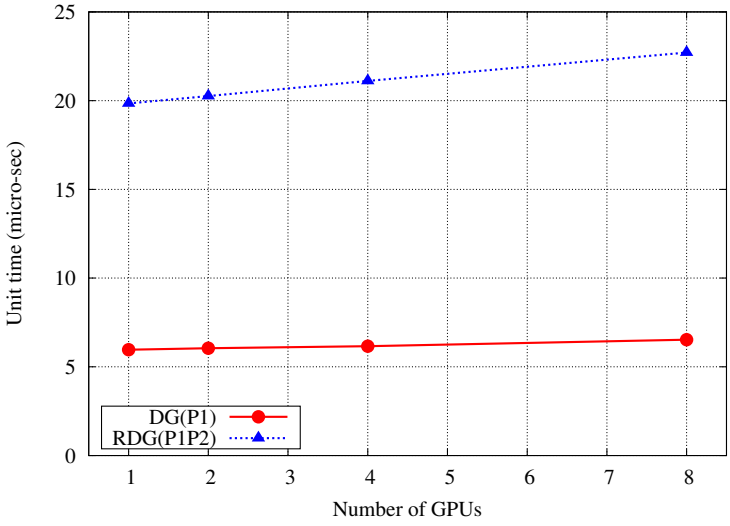
Table 9: Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2050 GPU cards for viscous flow past a sphere.

| | | | Storage (GB) | | Unit time (ms) | | Parallel efficiency | |
|---|---|---|---|---|---|---|---|---|
| Grids | Elements | GPU's | P1 | P1P2 | P1 | P1P2 | P1 | P1P2 |
| Level 1 | $500,095$ | $\times 1$ | 1.9 | 2.5 | 5.96 | 19.84 | – | – |
| Level 2 | $1,000,103$ | $\times 2$ | 3.8 | 4.9 | 6.05 | 20.26 | 98.5% | 97.9% |
| Level 3 | $2,000,051$ | $\times 4$ | 7.6 | 9.9 | 6.16 | 21.11 | 96.6% | 93.6% |
| Level 4 | $4,000,227$ | $\times 8$ | 15.2 | 19.8 | 6.53 | 22.71 | 90.4% | 85.5% |

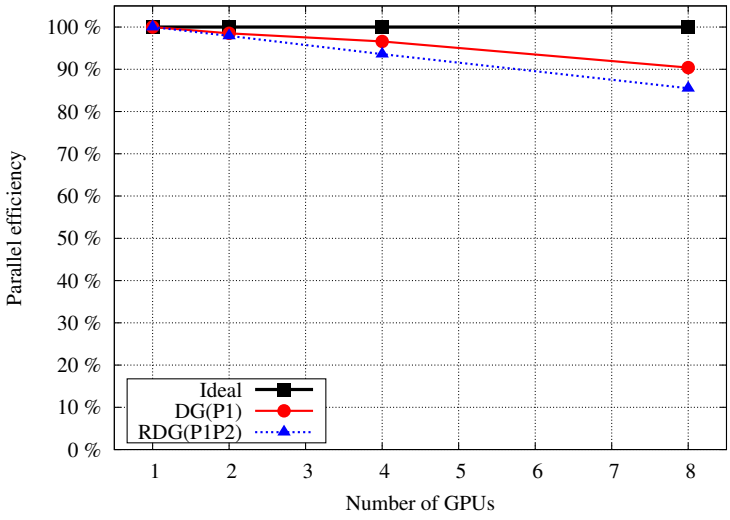
### C. Quasi-2D lid driven square cavity

A quasi-2D lid driven square cavity laminar flow at a Reynolds numbers of $Re = 10,000$ is considered in this numerical experiment. The cavity dimensions are 1 unit in the $x$ and $y$ directions, and 0.1 unit in the $z$ direction. First the computation is conducted on four Nvidia Tesla C2050 GPU cards to verify if the GPU code can deliver the identical numerical solutions to those by the equivalent CPU code. A sparse hexahedral grid which was used in Reference,[35] is used in the verification test as shown in Fig. 6(a). This grid consists of $32 \times 32 \times 2$ grid points, which has only one element in the span-wise $z$ direction. 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. On the front and back walls, a symmetric boundary condition is prescribed. The computation is started with the velocity field at rest without perturbation, and terminated at a sufficiently large total number of time iterations, ensuring that a steady-state flow field is reached. The computed velocity streamtraces obtained by the GPU code is displayed in 6(b), demonstrating the ability of the RDG(P1P2) method to accurately resolve all the major vortices on this sparse grid. Fig. 7 displays the profiles of the computed normalized velocity components $u/u_B$ and $v/u_B$ by DG(P1) and RDG(P1P2) that are plotted along the $y$ and $x$ center-lines respectively. The profiles by a second-order compressible finite volume solver based on a WENO reconstruction,[30] namely RDG(P0P1) in our RDG(P$n$P$m$) framework, is also presented. As one can observe, only the profiles by RDG(P1P2) matched well with the classical reference data by Ghia *et al.*,[13] clearly demonstrating the superior accuracy of the RDG(P1P2) method in the case of high Reynolds numbers and very sparse grid resolution. Above all, a `diff` check with an absolute error tolerance of $1.0 \times 10^{-12}$ indicates that the GPU code and the CPU code produced the identical solution data in this verification test.

Secondly a weak scaling test is carried out on a sequence of four successively refined hexahedral grids, which contain half million ($1000 \times 500$), one million ($1000 \times 1000$), two million ($1000 \times 2000$), and four million ($1000 \times 4000$) elements, respectively. These four grids correspond to the use of one, two, four, and eight Nvidia Tesla C2050 GPU cards respectively, ensuring a fixed problem size per GPU card. The total number of time iterations is set to be $10,000$ for all of these four grids. The detailed timing measurements are presented in Table 10, showing the statistics of unit running time and parallel efficiency obtained on each grid. In addition, Figs. 8(a) and 8(b) displayed the variations of the unit running time and parallel efficiency

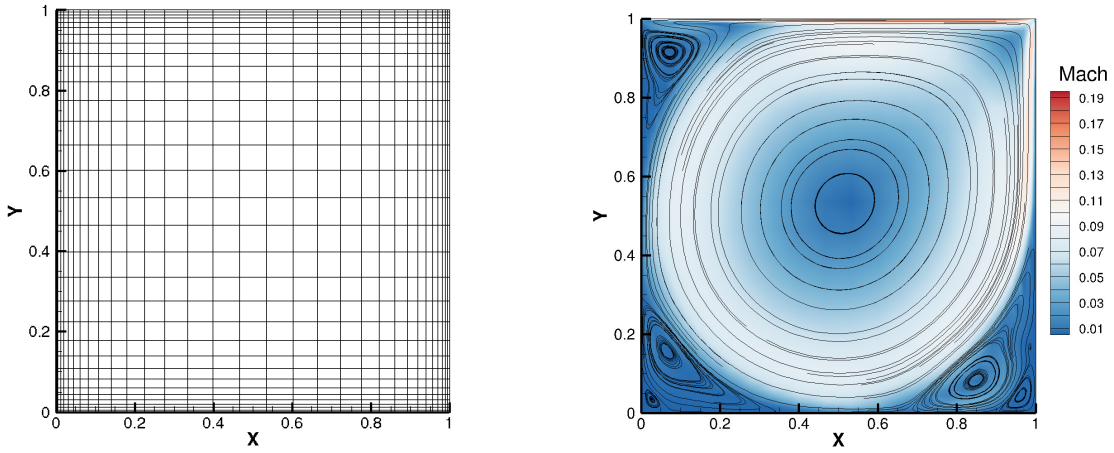



(a)




(b)

Figure 5: Plot of the timing measurements for viscous flow past a sphere with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards; (b) Parallel efficiency versus the number of GPU cards.




(a) (b)

Figure 6: A quasi-2D lid driven square cavity flow at a lid velocity of $\mathbf{V}_B = (0.2, 0, 0)$ and a Reynolds number of $Re = 10,000$: (a) The sparse hexahedral grid ($32 \times 32 \times 2$ grid points) used in the verification test. (b) The computed streamtraces obtained by RDG(P1P2) on the sparse grid.

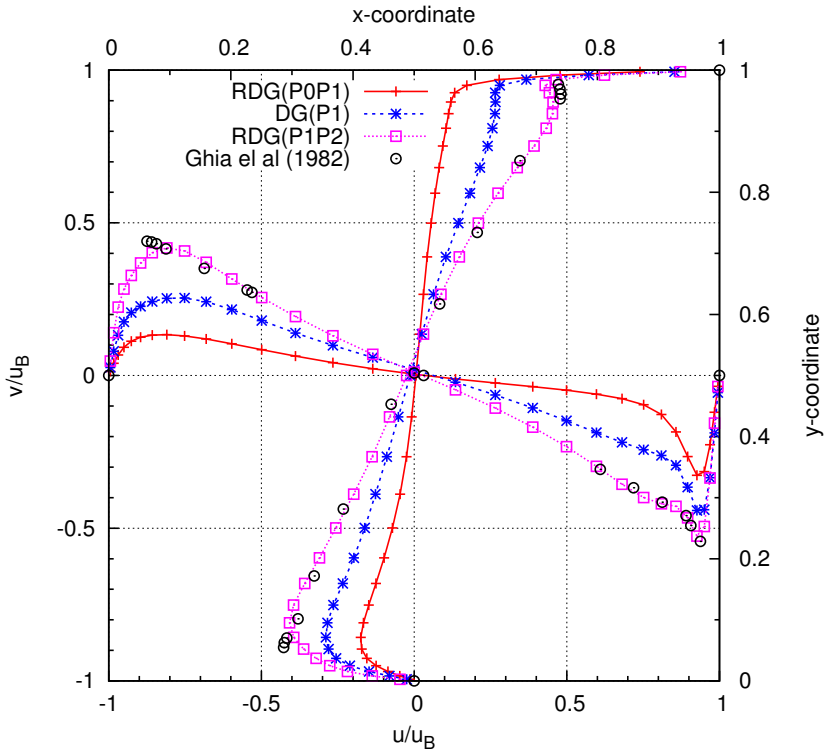



Figure 7: Profiles of the normalized velocity components $u/u_B$ and $v/u_B$ on a sparse hexahedral grid ($32 \times 32 \times 2$ grid points) for a quasi-2D lid driven square cavity at a lid velocity of $\mathbf{V}_B = (0.2, 0, 0)$, and a Reynolds number of $Re = 10,000$.

with respect to the number of GPUs, demonstrating that both the DG(P1) and RDG(P1P2) methods have achieved excellent parallel efficiency in the case of eight GPUs, being 97.0% and 97.3% respectively. Such a high scalability as obtained in this test case is mainly attributed to the use of hexahedral grids, which renders better load-balanced grid partitions and renumbered face groups than tetrahedral grids. Above all, this parallel RDG solver has exhibited strong scalability on multiple GPUs for computing viscous flow problems on hexahedral grids.

Table 10: Timing measurements of weak scaling obtained on a cluster of Nvidia Tesla C2070 GPU cards for a quasi-2D lid driven square cavity.

| | | | Storage (GB) | | Unit time (ms) | | Parallel efficiency | |
|---|---|---|---|---|---|---|---|---|
| Grids | Elements | GPU's | P1 | P1P2 | P1 | P1P2 | P1 | P1P2 |
| Level 1 | 500,000 | ×1 | 2.3 | 3.0 | 9.40 | 15.04 | – | – |
| Level 2 | 1,000,000 | ×2 | 4.6 | 6.1 | 9.54 | 15.33 | 98.5% | 98.1% |
| Level 3 | 2,000,000 | ×4 | 9.2 | 12.2 | 9.63 | 15.72 | 97.6% | 95.7% |
| Level 4 | 4,000,000 | ×8 | 19.6 | 24.4 | 9.68 | 16.47 | 97.0% | 91.3% |

## VI. Conclusions

A multi-GPU accelerated, reconstructed discontinuous Galerkin method RDG(P1P2) has been developed based on the OpenACC directives for the solution of compressible flows on 3D hybrid grids. A remarkable design feature in the present scheme is that it requires minimum intrusion and algorithm alteration to an existing CPU code, and renders an efficient approach to upgrading a legacy solver with the GPU-computing capability without compromising its cross-platform portability and compatibility with the mainstream compilers. Communication between the GPUs is achieved via the host CPUs and synchronized based on the MPI library. A face renumbering and grouping algorithm is used to eliminate memory contention of vectorized computing over the face loops on each individual GPU. A series of inviscid and viscous flow problems have been presented for the verification and scaling test, demonstrating excellent scalability of the resulting GPU code. The numerical results indicate that this RDG(P1P2) method is an efficient high-order DG method for scalable computing on GPU clusters. In addition to the Nvidia GPGPUs, future work will be focused on the completion of a full portability study on the AMD and Intel GPGPUs for the developed flow solver.

## Acknowledgments

The authors would 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 serve as the technical monitors.

## References

[1]V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, and K. C. Giannakoglou. Unsteady cfd computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *International Journal for Numerical Methods in Fluids*, 67(2):232–246, 2011.

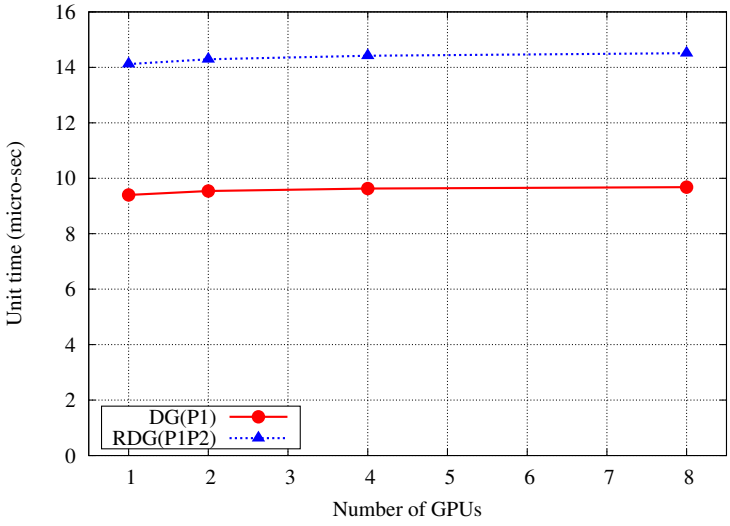
[2]F. Bassi and S. Rebay. Discontinuous Galerkin Solution of the Reynolds-Averaged Navier-Stokes and $\kappa$-$\omega$ Turbulence Model Equations. *Computers & Fluids*, 34(4-5):507–540, 2005.

[3]P. Batten, M. A. Leschziner, and U. C. Goldberg. Average-State Jacobians and Implicit Methods for Compressible Viscous and Turbulent Flows. *Journal of Computational Physics*, 137(1):38–78, 1997.

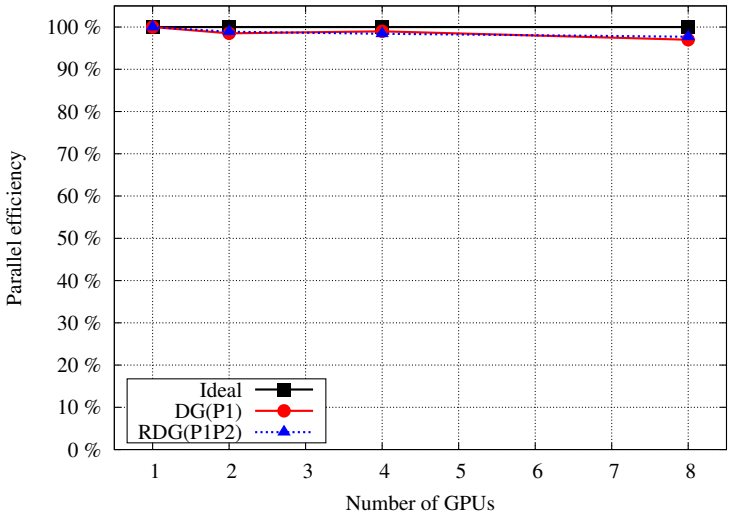
[4]T. Brandvik and G. Pullan. Acceleration of a two-dimensional euler flow solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 221(12):1745–1748, 2007.

[5]T. Brandvik and G. Pullan. Acceleration of a 3d euler solver using commodity graphics hardware. In *46th AIAA aerospace sciences meeting and exhibit*, pages 2008–607, 2008.

[6]B. Cockburn, S. Hou, and C. W. Shu. TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for conservation laws IV: the Multidimensional Case. *Journal of Mathematical Physics*, 55:545–581, 1990.




(a)




(b)

Figure 8: Plot of the timing measurements for a quasi-2D lid driven square cavity with a fixed problem size (approximately half million elements) per GPU card (Nvidia Tesla C2050): (a) Unit running time versus the number of GPU cards. (b) Parallel efficiency versus the number of GPU cards.

[7]B. Cockburn and C. W. Shu. The Runge-Kutta Discontinuous Galerkin Method for conservation laws V: Multidimensional System. *Journal of Computational Physics*, 141:199–224, 1998.

[8]J. Cohen and M. J. Molemaker. A fast double precision cfd code using cuda. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, 2009.

[9]A. Corrigan, F. Camelli, R. Löhner, and F. Mut. Porting of an edge-based cfd solver to gpus. In *48th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, 2010.

[10]A. Corrigan, F. Camelli, R. Löhner, and F. Mut. Semi-automatic porting of a large-scale fortran cfd code to gpus. *International Journal for Numerical Methods in Fluids*, 69(2):314–331, 2012.

[11]A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, 2011.

[12]E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a gpu. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

[13]U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48(3):387–411, 1982.

[14]D. Goddeke, S. HM Buijssen, H. Wobker, and S. Turek. Gpu acceleration of an unmodified parallel finite element navier-stokes solver. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 12–21. IEEE, 2009.

[15]D. A. Jacobsen, J. C. Thibault, and I. Senocak. An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters. In *48th AIAA Aerospace Sciences Meeting and Exhibit*, volume 16, 2010.

[16]D. C. Jespersen. Acceleration of a cfd code with a gpu. *Scientific Programming*, 18(3):193–201, 2010.

[17]H. Jin, M. Kellogg, and P. Mehrotra. Using compiler directives for accelerating CFD applications on GPUs. In *OpenMP in a Heterogeneous World*, pages 154–168. Springer, 2012.

[18]A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

[19]H. Luo, Y. Xia, S. Li, and R. Nourgaliev. A Hermite WENO Reconstruction-Based Discontinuous Galerkin Method for the Euler Equations on Tetrahedral grids. *Journal of Computational Physics*, 231(16):5489–5503, 2012.

[20]H. Luo, Y. Xia, S. Spiegel, R. Nourgaliev, and Z. Jiang. A reconstructed discontinuous Galerkin method based on a hierarchical WENO reconstruction for compressible flows on tetrahedral grids. *Journal of Computational Physics*, 236:477–492, 2013.

[21]D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[22]J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[23]E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting, number AIAA*, volume 565, 2009.

[24]G. Shainer, A. Ayoub, P. Lui, T. Liu, M. Kagan, C. R. Trott, G. Scantlen, and P. S. Crozier. The development of Mellanox/NVIDIA GPUDirect over InfiniBanda new model for GPU to GPU communications. *Computer Science-Research and Development*, 26(3-4):267–273, 2011.

[25]J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[26]J. C. Thibault and I. Senocak. Cuda implementation of a navier-stokes solver on multi-gpu desktop platforms for incompressible flows. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, pages 2009–758, 2009.

[27]H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K Panda. MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters. *Computer Science-Research and Development*, 26(3-4):257–266, 2011.

[28]S. Wienke, P. Springer, C. Terboven, and D. Mey. Openaccfirst experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[29]Y. Xia, M. Frisbey, H. Luo, and R. Nourgaliev. A WENO Reconstruction-Based Discontinuous Galerkin Method for Compressible Flows on Hybrid Grids. *AIAA Paper*, 2013-0516, 2013.

[30]Y. Xia, X. Liu, and H. Luo. A Finite Volume Method Based on a WENO Reconstruction for Compressible Flows on Hybrid Grids. *AIAA Paper*, 2014-0939, 2014. http://arc.aiaa.org/doi/abs/10.2514/6.2014-0939.

[31]Y. Xia, H. Luo, M. Frisbey, and R. Nourgaliev. A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids. *Computers & Fluids*, 98:134–151, 2014. http://dx.doi.org/10.1016/j.compfluid.2014.01.023.

[32]Y. Xia, H. Luo, L. Luo, J. Edwards, J. Lou, and F. Mueller. OpenACC-based GPU Acceleration of a 3-D Unstructured Discontinuous Galerkin Method. *AIAA Paper*, 2014-1129, 2014. http://arc.aiaa.org/doi/abs/10.2514/6.2014-1129.

[33]Y. Xia, H. Luo, and R. Nourgaliev. An implicit Hermite WENO reconstruction-based discontinuous Galerkin method on tetrahedral grids. *Computers & Fluids*, 96:406–421, 2014. http://dx.doi.org/10.1016/j.compfluid.2014.02.027.

[34]Y. Xia, H. Luo, S. Spiegel, M. Frisbey, and R. Nourgaliev. A Parallel, Implicit Reconstruction-Based Hermite-WENO Discontinuous Galerkin Method for the Compressible Flows on 3D Arbitrary Grids. *AIAA Paper*, submitted, in process, 2013.

[35]Y. Xia, H. Luo, C. Wang, and R. Nourgaliev. Implicit Large Eddy Simulation of Turbulent Flows by a Reconstructed Discontinuous Galerkin Method. *AIAA Paper*, 2014-0224, 2014. http://arc.aiaa.org/doi/abs/10.2514/6.2014-0224.

[36]B. Zimmerman, Z. Wang, and M. Visbal. High-Order Spectral Difference: Verification and Acceleration using GPU Computing. 2013-2491, 2013.