OpenACC directive-based GPU acceleration of an implicit reconstructed discontinuous Galerkin method for compressible flows on 3D unstructured grids

CONFERENCE PAPER · JANUARY 2016

DOI: 10.2514/6.2016-1815

6 AUTHORS, INCLUDING:

Yidong Xia

Idaho National Laboratory

30 PUBLICATIONS **75** CITATIONS

SEE PROFILE

Lixiang Luo

North Carolina State University

23 PUBLICATIONS **61** CITATIONS

SEE PROFILE

Hong Luo

Sichuan Normal University

97 PUBLICATIONS **1,377** CITATIONS

SEE PROFILE

AIAA SciTech
4-8 January 2016, San Diego, California, USA
54th AIAA Aerospace Sciences Meeting

OpenACC directive-based GPU acceleration of an implicit reconstructed discontinuous Galerkin method for compressible flows on 3D unstructured grids

Jialin Lou [*1], Yidong Xia [†2], Lixiang Luo [‡1], Hong Luo [§1], Jack Edwards [¶1], and Frank Mueller [‖1]

[1] *North Carolina State University, Raleigh, NC 27695, United States*
[2] *Idaho National Laboratory, Idaho Falls, ID 83415, United States*

Despite of the increasing popularity of OpenACC directive-based acceleration for computational fluid dynamics (CFD) codes using the general-purpose graphics processing units (GPGPUs), an efficient implicit algorithm for high-order method on unstructured grids is still a relatively unexplored area. This is mainly due to the fact that, the capacity of local cache memory of a top-notch GPGPU is still far behind a common CPU. Thus many state-of-the-art preconditioning algorithms (e.g. the Symmetric Gauss-Seidel (SGS) and Lower Upper-Symmetric Gauss-Seidel (LU-SGS)), in which the matrix and strongly inherent data dependent operations are heavily involved, become extremely inefficient because of the local cache memory bound, when simply ported onto GPGPUs. In the present study, an efficient implicit algorithm for a GPGPU accelerated reconstructed discontinuous Galerkin (DG) CFD code is introduced and assessed for the solution of the Euler equations on unstructured grids. The block matrix operations are refined to element level. A Gauss-Jordan elimination based matrix inversion algorithm is adopted to optimize the performance on GPU platform. For SGS-type linear solver/preconditioner, a straightforward element reordering algorithm is employed to eliminate data dependency. As a result, the developed algorithm is implemented on GPGPU to accelerate a high-order implicit reconstructed discontinuous Galerkin (rDG) method as a compressible flow solver on 3D unstructured grids. Several numerical tests are carried out to obtain the speed up factor as well as the parallel efficiency, which indicates that the presented algorithm is able to offer low-overhead concurrent CFD simulation on unstructured grids on NVIDIA GPGPUs.

I. Introduction

Nowadays, with increasing attention in science and engineering filed, the general-purpose graphics processing unit (GPGPU[24]) technology offers a new opportunity to significantly accelerate the CPU-based code by offloading compute-intensive portions of the application to the GPU, while the remainder of the computer program still runs on the CPU, which also make it expected to be a major compute unit in the near future. Computational Fluid Dynamics (CFD), a branch of mechanics that applies numerical way to solve fluid dynamics problem, has been one of the most significant applications on supercomputers. The presence of GPGPU could outperform the traditional CPU based parallel computing and therefore to meet the needs to solving complex simulation CFD problem.

As a popular parallel programming model and and platform in GPGPU technology, NVIDIA's CUDA application programming interface (API) and CUDA-enabled accelerators has drawn many researchers' attention. Therefore, people have put effort in the investigation and development GPU-accelerated CFD solvers with the CUDA technology.[1,4–8,11–13,23,25,30] As a matter of fact, the numerical methods range from

[*] Ph.D Student at Department of Mechanical and Aerospace Engineering, Student Member AIAA.
[†] Research Scientist at Department of Energy Resource Recovery and Sustainability.
[‡] Postdoctoral Research Associate at Department of Mechanical and Aerospace Engineering, Member AIAA.
[§] Professor at Department of Mechanical and Aerospace Engineering, Associate Fellow AIAA.
[¶] Professor at Department of Mechanical and Aerospace Engineering, Associate Fellow AIAA.
[‖] Professor at Department of Computer Science.

the finite difference methods (FDMs), spectral difference methods (SDMs), finite volume methods (FVMs), discontinuous Galerkin methods (DGMs) to Lattice Boltzmann method (LBMs). For instance, Elsen *et al.*[10] reported a 3D high-order FDM solver for large calculation on multi-block structured grids; Klöckner *et al.*[15] developed a 3D unstructured high-order nodal DGM solver for the Maxwell's equations; Corrigan *et al.*[9] proposed a 3D FVM solver for compressible inviscid flows on unstructured tetrahedral grids; Zimmerman *et al.*[39] presented an SDM solver for the Navier-Stokes equations on unstructured hexahedral grids.

For either production level or research level, people would prefer to maintain multi-platform compatibility at a minimum extra cost in time and effort. As for the most popular programming model in GPU, that is, CUDA technology, people would either need to start a brand new code design or extend an existing CPU code to GPU platform to develop a GPU accelerated CFD solver. The former one is often the case of fundamental study of a numerical model on GPU computing while the latter one requires applying NVIDIA CUDA model to a legacy CFD code, which is not an easy job since the developer has to define an explicit layout of the threads on the GPU (numbers of blocks, numbers of threads) for each kernel function.[14] Nevertheless, adopting CUDA might spell almost a brand new design and long-term project, and a constraint to the CUDA-enabled devices, thus to lose the code portability on other platforms. Therefore, some alternatives come into play including OpenCL:[29] the currently dominant open GPGPU programming model (but dropped from further discussion since it does not support Fortran); and OpenACC:[31] a new open parallel programming standard based on directives.

Similar to OpenMP, OpenACC is a directive based programing model. Therefore, what developers need to do is simply annotate their code to identify the areas that should be accelerated by wrapping with the OpenACC directives and some runtime library routines, instead of taking the huge effort to change the original algorithms as to accommodate the code to a specific GPU architecture and compiler. In that case, 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, e.g., NVIDIA, AMD and Intel accelerators. Nevertheless, as for some cutting-edge features, OpenACC still lags behind CUDA due to vendors' distribution plan (note that Nvidia is among the OpenACC's main supporters). But still, OpenACC manages to offer a promising approach to minimize the effort to extend the existing legacy CFD codes while maintaining multi-platform and multi-vendor compatibility, and thus to become an attractive parallel programming model.

The objective of the effort discussed in the present work is to develop a OpenACC directive-based implicit algorithm for a reconstructed discontinuous Galerkin method. This work is based on a class of reconstruction-based rDG(PnPm) methods,[20,21,32,36–38] which are recently developed in order to improve the overall performance of the underlying standard DG(Pn) methods without significant extra costs in terms of computing time and storage requirement. Due to the fact that OpenACC could offer multi-platform/complier support with minor effort to code directives, it has been employed to partially upgrade a legacy CFD solver with the GPU computing capacity. A face renumbering and grouping algorithm is used to eliminate "race condition" in face-based flux calculation on GPU vectorization. Therefore, part of the solution modules in a well verified and validated rDG flow solver have already been upgraded with the capability of both single-GPU and multiple-GPU computing based on the OpenACC directives in our prior work.[17,33–35]

A typical GPU has hundreds or even thousands of computation cores. However, compared with a typical CPU core, the one on GPU card would has much less computation power and local cache memory. Therefore, for optimal performance on GPU, the algorithm should be divided into smaller units, thus occupying more cores with the same amount of work. In addition, the amount of data each core processes should be kept as small as possible. Therefore, the fine granularity algorithm, with the two above-mentioned benefits, is able to carry out high computation efficiency on GPGPU. Note that the latter aspect is particularly important for solving a block-sparse system, which consists of a large amount of square sub-matrices of an identical size. On each sub-matrix, if the operation is mapped to one GPU core, the algorithm easily becomes heavily memory-bounded, especially when the dimension of the sub-matrix is large, which is the comman case for high order method, resulting in serious performance penalty. In other words, the key to achieve higher speed up factor on GPU platform is fine granularity. On the other hand, fine granularity usually introduces more synchronization and memory access overhead. Compared with other open GPGPU programing model like OpenCL, OpenACC would suffer a more expensive overhead from kernel launches. Thus, the algorithm designers need to weigh costs and benefits to make a balanced choice.

Nevertheless, it is generally difficult to port the implicit algorithm to GPU platform. First of all, the size of sub-matrices would cause the memory-bounded issue, especially for higher order method. Secondly, it is

not straightforward to employ some popular liner solver or preconditioner like Symmetric Gauss-Seidel (SGS) or Lower Upper-Symmetric Gauss-Seidel (LU-SGS) due to the inherent data dependency. Additionally, the iterative solver like Generalized Minimal Residual (GMRES) method would require additional storage for some auxiliary arrays, which would bring challenge to GPU computing since the local cache memory of GPGPU is limited.

This paper aims to overcome several above-mentioned difficulties to thread the implicit algorithm on GPGPU based on OpenACC directives. First, we would need to refine the matrix operations for block sub-matrices at the matrix element level to achieve optimal performance on GPGPU. For matrix-matrix and matrix-vector multiplications, the fine-grained algorithms are straightforward, while the fine-grained algorithm for matrix inversion is particularly elusive. It is accomplished by a parallel in-place Gauss-Jordan elimination.[22] Meanwhile, the LU-SGS or SGS solver/preconditioner is easy to vectorized by using hyper-planes $i + j + k =$ const for structured mesh. However, when it comes to unstructured grids, an element renumbering algorithm is carried out to generate hyperplanes, each consisting of data-independent elements.

The developed method is used to compute the compressible flows for a variety of inviscid test problems on unstructured grids. Speed-up factors that achieved by strong scaling test, which compare the computing time of the OpenACC program on an NVIDIA Tesla K20c GPU and that of the equivalent MPI program on one single core and full sixteen cores of an AMD Opteron-6128 CPU indicate that implicit reconstructed discontinuous Galerkin method is a cost-effective high-order scheme on OpenACC-based GPU platform. If more than one GPU is used, the grid partitioning would be performed and loaded equally on each device. Communication between the GPUs is done with the help of the host-based MPI. The weak scaling test would be carried out to test the parallel efficiency by varying the number of NVIDIA Tesla C2050 for a fixed problem size per GPU card.

The outline of the rest of this paper is organized as follows. The governing equations are briefly introduced in Section II. In Section III, the discontinuous Galerkin spatial discretization is described. In Section IV, the idea of implicit reconstructed discontinuous Galerkin method is given. In Section V, the keynotes of porting an implicit reconstructed discontinuous Galerkin flow solver to GPU based on the OpenACC directives is discussed in detail. In Section VI, a series of numerical test cases are presented. Finally the concluding remarks and some discussions on further improvements are given in Section VII.

II. Governing Equations

The Euler equations governing the unsteady compressible inviscid flows can be expressed as

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_k(\mathbf{U})}{\partial x_k} = 0 \tag{1}$$

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

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

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 for air or diatomic perfect gas.

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 Ω is subdivided into a collection of non-overlapping arbitrary elements Ω_e in 3D, and then introduce the following broken Sobolev space V_h^p

$$V_h^p = \left\{ v_h \in \left[L^2(\Omega)\right]^m : v_h|_{\Omega_e} \in \left[V_p^m\right] \forall \Omega_e \in \Omega \right\} \tag{4}$$

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 Ω_e, and then performing an integration by parts: find $\mathbf{U}_h \in V_h^p$ such as

$$\frac{d}{dt}\int_{\Omega_e} \mathbf{U}_h \mathbf{W}_h d\Omega + \int_{\Gamma_e} \mathbf{F}_k \mathbf{n}_k \mathbf{W}_h d\Gamma - \int_{\Omega_e} \mathbf{F}_k \frac{\partial \mathbf{W}_h}{\partial x_k} d\Omega = 0, \quad \forall \mathbf{W}_h \in V_h^p \tag{5}$$

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 Γ_e: the boundary of Ω_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,

$$\frac{d}{dt}\int_{\Omega_e} \mathbf{U}_h B_i d\Omega + \int_{\Gamma_e} \mathbf{F}_k \mathbf{n}_k B_i d\Gamma - \int_{\Omega_e} \mathbf{F}_k \frac{\partial B_i}{\partial x_k} d\Omega = 0, \quad 1 \leq i \leq N \tag{6}$$

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] is used for evaluating the inviscid fluxes.

By moving the second and third terms to the right-hand-side (r.h.s.) in Eq. 6, we will arrive at a system of ordinary differential equations (ODEs) in time, which can be written in semi-discrete form as

$$\boldsymbol{M}\frac{d\mathbf{U}}{dt} = \mathbf{R}(\mathbf{U}) \tag{7}$$

where $\boldsymbol{M}$ is the mass matrix and $\mathbf{R}$ is the residual vector. The present work employs an GPU accelerated implicit rDG method, which is a third-order, WENO reconstructed scheme.

IV. Euler Implicit time integration

Euler implicit time integration would rewrite the semi-discrete system of ordinary differential equations, i.e., Eq. 7 as

$$\mathbf{M}_i \frac{\triangle \mathbf{U}_i^n}{\triangle t} = \mathbf{R}_i^{n+1} \tag{8}$$

where the $\triangle t$ is the time increment, and $\triangle \mathbf{U}^n$ the difference of unknown vector between levels n and $n+1$. The above equation can be linearized in time as

$$\mathbf{M}_i \frac{\triangle \mathbf{U}_i^n}{\triangle t} = \mathbf{R}_i^n + \frac{\partial \mathbf{R}_i^n}{\partial \mathbf{U}} \triangle \mathbf{U}_i \tag{9}$$

Now, we could solve a linear system as

$$\left(\frac{\mathbf{M}_i}{\triangle t}\mathbf{I} - \frac{\partial \mathbf{R}^n}{\partial \mathbf{U}}\right) \triangle \mathbf{U}_i = \mathbf{R} \tag{10}$$

The following simplified flux function is used to obtain the left-hand-side Jacobian matrix

$$\mathbf{R}_i = \sum_j \frac{1}{2}[\mathbf{F}(\mathbf{U}_i, \mathbf{n}_{ij}) + \mathbf{F}(\mathbf{U}_j, \mathbf{n}_{ij}) - |\lambda_{ij}|(\mathbf{U}_j - \mathbf{U}_i)]|\mathbf{s}_{ij}| \tag{11}$$

where

$$|\lambda_{ij}| = |\mathbf{V}_{ij} \cdot \mathbf{n}_{ij}| + C_{ij} + \frac{\mu_{ij}}{\rho_{ij}|\mathbf{x}_j - \mathbf{x}_i|} \tag{12}$$

where $\mathbf{s}_{ij}$ is the area vector normal to the control volume interface associated with the element ij, $\mathbf{n}_{ij} = \mathbf{s}_{ij}/|\mathbf{s}_{ij}|$ its unit vector in the direction $\mathbf{s}_{ij}$, C_{ij} the speed of sound, and the summation is over all neighboring

interfaces. Using an elemental based data structure, the left-hand-side Jacobian matrix is stored in upper, lower, and diagonal forms, which can be expressed as

$$U_{ij} = -\frac{1}{2}(J(\mathbf{U}_j, \mathbf{n}_{ij}) - |\lambda_{ij}\mathbf{I}|)|s_{ij}| \tag{13}$$

$$L_{ij} = \frac{1}{2}(J(\mathbf{U}_i, \mathbf{n}_{ij}) + |\lambda_{ij}\mathbf{I}|)|s_{ij}| \tag{14}$$

$$D_{ij} = \frac{M}{\triangle t}\mathbf{I} - \sum_j \frac{1}{2}(J(\mathbf{U}_i, \mathbf{n}_{ij}) + |\lambda_{ij}\mathbf{I}|)|s_{ij}| \tag{15}$$

Eq. 9 represents a system of linear simultaneous algebraic equations. The most widely used methods to solve this linear system are iterative solution methods and approximate factorization methods. The LU-SGS and SGS approximate factorization method are attractive due to their good stability properties and competitive computational cost. Although those methods are more efficient than their explicit counterpart, a significant number of time steps are still required to achieve the steady state solution, because of the nature of the approximation factorization schemes. One way to speed up the convergence is to use iterative methods like GMRES. It has been shown that GMRES with LU-SGS as its preconditioner method (GMRES+LU-SGS) would result in very good convergence for unstructured meshes.[18,19,27] In that case, the preconditioner must be very fast, and at the same time it should resemble the original Jacobian matrix as close as possible. Preconditioning will be cost-effective only if the additional computational work incurred for each sub-iteration is compensated for by a reduction in the total number of iterations to convergence. Thus, even a moderate inefficiency in parallelization of the preconditioner can be critical. The focus of this study would be on LU-SGS and SGS solver/precondtioner while the effort on threading GMRES algorithm to GPGPU is currently being worked on. Our final goal is not to solve the system entirely by LU-SGS or SGS approximate factorization but rather implement GMRES+LU-SGS scheme on GPGPU based on OpenACC directives.

A. The LU-SGS approximate factorization

As we shown above, the left-hand side Jacobian matrix A is stored in lower, upper, and diagonal forms, which can also be expressed as

$$A = \left(\frac{M}{\triangle t}\mathbf{I} - \frac{\partial \mathbf{R^n}}{\partial \mathbf{U}}\right) = L + U + D = (D + L)D^{-1}(D + U) - LD^{-1}U \tag{16}$$

Neglecting the last term, the system can be solved in the two steps. First, a lower (forward) sweep:

$$(D + L)\triangle\mathbf{U}^* = \mathbf{R} \tag{17}$$

and second, an upper (backward) sweep:

$$(D + U)\triangle\mathbf{U} = D\triangle\mathbf{U}^* \tag{18}$$

B. Symmetric Gauss-Seidel relaxation

As for SGS, we would zero the $\triangle\mathbf{U}$ array first.

$$\triangle\mathbf{U}^0 = 0 \tag{19}$$

Then the $k_{\max}$ subiteration are made using forward sweep:

$$(D + L)\triangle\mathbf{U}^{k+1/2} = \mathbf{R} - U\triangle\mathbf{U}^k \tag{20}$$

and second, an upper (backward) sweep:

$$(D + U)\triangle\mathbf{U}^{k+1} = \mathbf{R} - L\triangle\mathbf{U}^{k+1/2} \tag{21}$$

Note that for one subiteration ($k_{\max} = 1$), the SGS method is equivalent to the LU-SGS approximate factorization method. These sweeps can be vectorized by using special ordering technique,[28] but parallelization of LU-SGS or SGS algorithm is not straightforward due to inherent data dependencies.

V. OpenACC Implementation

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 Euler implicit time integration is utilized to update conservative variable vector. 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 <ACC> tag denotes an OpenACC acceleration-enabled region, and the <MPI> tag means that MPI routine calls are needed in the case that multiple devices are utilized. Clearly, two extra MPI routine calls are required for the rDG method compared with the standard DG method, due to the fact that the solution vector at the partition ghost elements need to be updated after each reconstruction scheme.

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

```
! Main time loop
DO itime = 1, ntime

   <ACC> Predict time-step size

   <ACC> Compute R.H.S. vector R
      !! P1P2 least-squares reconstruction
      <ACC> IF (nreco > 0) CALL reconstruction_ls(...)

      !! data exchange for partition ghost cells
      <MPI> IF (nprcs > 1) CALL exchange(...)

      !! WENO reconstruction
      <ACC> IF (nreco > 0) CALL reconstruction_weno(...)

      !! data exchange for partition ghost cells
      <MPI> IF (nprcs > 1) CALL exchange(...)
   <ACC> Compute L.H.S. matirx  A=M/dt-dR/dU

   <ACC> Solve the linear system Au=R
      !!Linear solver or iterative method
      ! Option 1: LU-SGS
      ! Option 2: SGS(k)
      ! Option 3: LUSGS + GMRES (not on GPU yet)

    <ACC> update solution vector
      !! data exchange for partition ghost cells
      <MPI> IF (nprcs > 1) CALL exchange(...)

ENDDO
```

One would need to compute R.H.S. and L.H.S. and thus to apply the linear solver like LU-SGS, SGS(k), or GMRES[2,26] to solve the linear system. For the residual computing, the most expensive workload includes both the reconstruction of the second derivatives and the accumulation of the right hand side residual vector in Eq. 9. 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 of OpenMP. The example shown in Table 2 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 2 without the need to modify the original code.

For implicit algorithm, one would need to obtain the inversion of each sub-matrix. We have tested both iterative methods like Jacobi method, Gauss-Seidel method and also direct methods like Gauss-Jordan elimination (GJE). The iterative methods can be easily ported to GPU platform with the speedup factor up to 69.02 according to our test. However, the great performance of iterative methods comes with a cost. It can only work under the circumstance that the matrix is diagonal dominant, which is not the common case

Table 2: 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
```


for CFD simulation on unstructured grids. However, all the direct inversion algorithms invariably involve recursive process, and would be inefficient if simply ported it to GPGPU. In our study, we would adopt GJE based direct inversion algorithm,[22] whose sequential algorithm is given in Table 3. Since this method is highly compact and does not require any extra storage, it would be particularly efficient on GPUs if we refine the algorithm to element level. The outer r-loop corresponds to the index of the row being transformed in GJE. Pivoting is generally not required, but it can be included if necessary. Although GJE is not the fastest method to compute matrix inversion, it is rather parallelizable. Also, it can be performed in-place, avoiding any thread-private arrays. Both factors are highly beneficial for GPU implementations. The algorithm must be written as three kernels due to the data dependencies, and can be found in Table 4. These three kernels have different mappings between GPU threads and loop iterations. Conditions have been translated into index manipulations, avoiding idle threads for skipped iterations. Because the r-loop runs on a CPU, there are $3n$ kernel launches to complete the inversion.

Table 3: Gauss-Jordan elimination without pivoting for an $n \times n$ matrix

```
DO r = 1, n
  DOs u = 1, n
    IF (u .NE. r) THEN
       d(u,r)=-d(u,r)/d(r,r);d(r,u)=d(r,u)/d(r,r)
    ENDIF
  ENDDO
!
  DO u = 1, n
    DO v = 1,n
      IF ((u .NE. r) .AND. (v .NE. r)) THEN
        d(u,v)=d(u,v)+d(u,r)*d(r,v)*d(r,r)
      ENDIF
    ENDDO
  ENDDO
!
  d(r,r)=1/d(r,r)
ENDDO
```

However due to 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 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. 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 5, 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.[16] The number of groups

Table 4: ACC version of GJE without pivoting for *nelem* matrices of size $n \times n$

```
DO r = 1, n
!$acc kernels
!  kernel # 1
!$acc loop independent
  DO ip= 0,2(n-1)*nelem-1
    ie = ip/2/(n-1)
    a  = ip - ie*2*(n-1)+1; ie = ie + 1
    a  = a + (a >= r); a = a + (a >= a+n)
    t  = (a > r? 1:0)
    a  = a-t*n; b = t*(a-r)
    u = r + b; v = a - b;
    d(u,v,ie) = (1-2t)*d(u,v,ie)/d(r,r,ie)
  END DO
!  kernel # 2
!$acc loop independent
  DO ip = 0,(n-1)*(n-1)*nelem-1
    ie = ip/(n-1)/(n-1)
    t  = ip - ie*(n-1)*(n-1)
    u  = t/(n-1);v  = t - a*(n-1)+1
    ie = ie + 1; a = a + 1
    u = u + (u >= r); v = v + (v >= r)
    d(u,v,ie) = d(u,v,ie)+d(u,r,ie)*d(r,v,ie)*d(r,r,ie)
  END DO
!  kernel # 3
!$acc loop independent
  DO ie = 1, nelem
    d(r,r,ie) = 1/d(r,r,ie)
  ENDDO
!$acc end kernels
ENDDO
```

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. In a word, the face renumbering and grouping method 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.

Table 5: 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
      rhsel(*,*,iel) = rhsel(*,*,iel) - flux
      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
      rhsel(*,*,iel) = rhsel(*,*,iel) - flux
      rhsel(*,*,ier) = rhsel(*,*,ier) + flux
    ENDDO

  ENDDO
  !$acc end parallel

ENDDO
```

For structured grid, the LU-SGS or SGS algorithm is easy to vectorized, for the sweeps would be performed by using hyperplanes $i+j+k$ =const, where i, j, and k are the indices of a grid cell. Thus, forward sweep updates point (i,j,k) using already updated values at $(i-1,j,k)$, $(i,j-1,k)$, and $(i,j,k-1)$, whereas backward sweep uses cells $(i+1,j,k)$, $(i,j+1,k)$, and $(i,j,k+1)$. For the unstructured grid, we would refer to Sharov and Nakahashi's work[28] to renumber the elements and generate the hyperplanes, whose algorithm can be found in Table 6. In this case, the lower matrix L would be computed by the surrounded elements whose group marks are lower than the host element. Similarly, the upper matrix U would be computed by the neighbour elements with higher group marks. Therefore, the sweeps are performed by increasing or decreasing the group mark numbers. And in each group, the computation can be done concurrently, thus, the algorithm can be vertorized.

Table 6: Reordering algorithm vectorize LUSGS/SGS

```
1) Mark the starting element as hyperplane number 1.
   Set the current group mark Np as 1.
2) Add all the unmarked neighbour element of current group to Np+1.
   Set Np = Np + 1.
3) Repeat Step 2) until all the elements are marked.
4) Examine each hyperplane to make sure that any two of the elements in the same
   group would not be neighbour. Assign and updates the group marks if necessary.
5) Color each interface accordingly. That is, for forward sweeps, color the face
   by the larger ending group mark, and for backward sweeps, color the face by
   the smaller ending group mark.
```

Note that if this reordering algorithm is applied to structured grid with taking the corner element $(1,1,1)$ as a starting element, one would have each hyperplanes as $i+j+k$ =const. And this reordering is performed only once before the main time loop. Memory overhead of this method is minimal, since the only extra memory is to store the group marks, which make it suitable for GPU computing.

VI. 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 card used in the present work is NVIDIA Tesla K20c GPU containing 2496 multiprocessors and NVIDIA Telsa C2050 GPU containing 448 multiprocessors. The performance of the equivalent MPI-based parallel CPU program was measured on an AMD Opteron 6128 CPU containing 16 cores. 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 unit time T_{unit} is calculated as

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

where T_{run} refers to the time recorded for completing the entire time marching loop with a given number of time steps `Ntime`, excluding the start-up procedures, initial/end data translation, and solution file dumping.

A. Inviscid 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 in order to assess the performance of OpenACC-based GPU acceleration on implicit reconstructed DG method. The explicit counterpart can be found in authors' previous work.[33,35] Computation is conducted on a sequence of three successively refined tetrahedral grids, displayed in Figs. 1(a) – 1(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. And the computed surface pressure contours by implicit rDG(P1P2) are shown in Figs. 1(d) – 1(f).The quantitative measurement of the discretization errors has been done in the authors'

previous work,[33,35] which indicates that explicit rDG(P1P2) method achieved a formal order of accuracy of convergence, convincingly demonstrating the benefits of using the rDG method over its underlying baseline DG method.

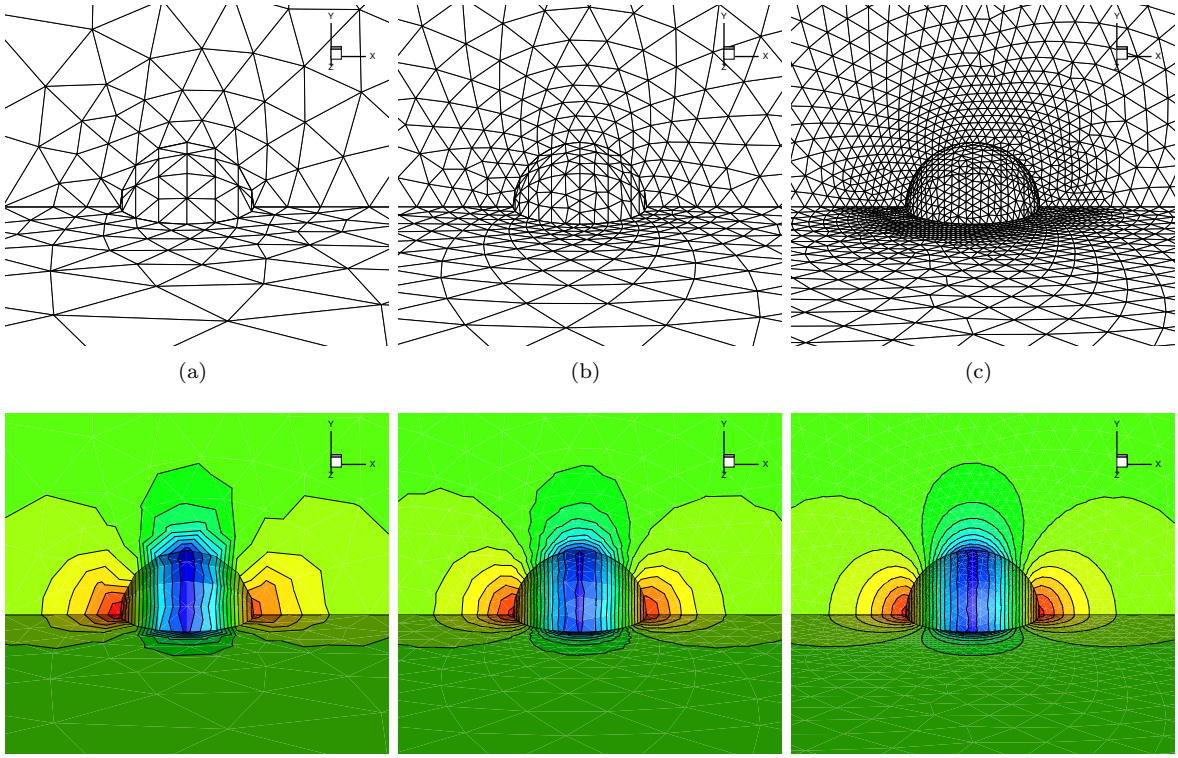


Figure 1: Subsonic flow past a sphere at a free-stream Mach number of $M_\infty = 0.5$: (a) – (c) Surface triangular meshes of the three successively refined tetrahedral grids; (d) – (f) Computed pressure contours obtained by implicit rDG(P1P2) on the surface meshes.

A strong scaling timing test is carried out on same sequence of four successively refined tetrahedral grids. The simulations are done by GPU-accelerated DG(P1) or rDG(P1P2) with SGS solver and original CPU code, so that we can see the effect of the GPU computing. A single NVIDIA Telsa K20c GPU card is used in this case, compared with single CPU serial computation. The detailed timing measurements are presented in Table 7, showing the statistics of unit running time. From the results we can see that GPU performs better when it comes to larger `Nelem`, and can achieve higher speedup factor when we use DGP1.

Table 7: Timing measurements of using implicit rDG methods for subsonic flow past a sphere.

	T_{unit} by implicit DG(P1)			T_{unit} by implicit rDG(P1P2)		
Nelem	GPU	CPU	Speedup	GPU	CPU	Speedup
2,426	41.63	74.60	1.79	44.52	81.62	1.83
16,467	15.18	77.43	5.10	18.22	85.89	4.71
124,706	11.18	80.99	7.12	14.03	88.85	6.33

For better illustration, a breakdown of the averaged one main loop run times is given in Figure 2. The result is based on running implicit rDG(P1P2) on the most fine mesh. From the breakdown, we can see the primary bottleneck for the current GPU implementation is LHS computing, which takes large percentage of the time while the speed up factor is only 3.29. The reason behind this is the memory bound issue,

since we have large amounts of memory accessed for a relatively small amount of computation. This part is implemented in a coarse-grained fashion, as the computation for each element is mapped to one GPU thread, making them highly memory-bound. To obtain full performance on GPUs, an ongoing effort is being undertaken to develop fine-grained versions of the related subroutines.

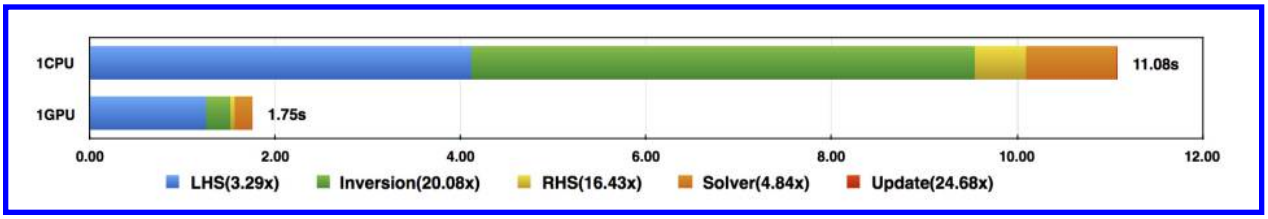


Figure 2: Averaged one main time loop run times (in seconds) for running implicit rDG(P1P2) on the finest mesh for the inviscid sphere case.

Next, a weak scaling test is carried out on a sequence of four successively refined tetrahedral grids. 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 8, showing the statistics of unit running time and parallel efficiency obtained on each grid. Both the DG(P1) and rDG(P1P2) methods have achieved good parallel efficiency in the case of eight GPUs, being 89.7% and 88.3% 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.

Table 8: Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for inviscid subsonic flow past a sphere.

			Unit time (ms)		Parallel efficiency	
Grids	Elements	GPU's	P1	P1P2	P1	P1P2
Level 1	62,481	×1	26.25	31.69	–	–
Level 2	124,706	×2	28.48	34.00	92.2%	93.2%
Level 3	249,945	×4	28.80	34.89	91.1%	90.8%
Level 4	501,972	×8	29.28	35.87	89.7%	88.3%

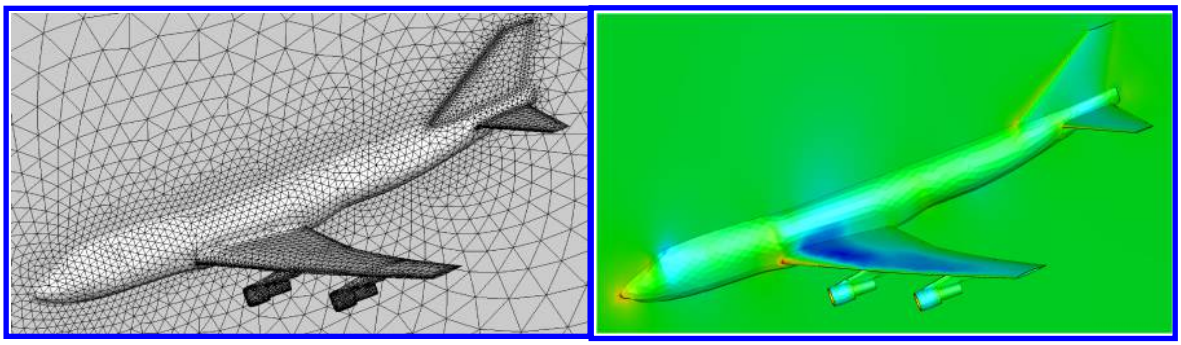
B. Transonic Flow over a Boeing 747 Aircraft

In the second test case, we choose a transonic flow pasting a Boeing 747 aircraft at a free stream Mach number of $M_\infty = 0.85$, and an angle of attack of $\alpha = 2°$. This case could test the ability of computing complex geometric configurations by a OpenACC-based GPU program. The configuration of Boeing 747 includes the fuselage, wing, horizontal and vertical tails, under-wing pylons, and flow-through engine nacelle. The grids we are using here are tetrahedron grid. Similarly, we only model half of the aircraft because of the symmetry of the problem. The grid is shown in Fig. 3(a). The computed pressure contours obtained by implicit rDG(P1P2) solution in the flow field are shown in Figs.3(b). Again, the explicit part can be found in authors' previous work.[35]

First, we would conduct a strong scaling test on a NVIDIA Telsa K20c GPU, with the timing measurement obtained by implicit rDG method presented in Table 9. Speedup factor of up to 8.23 can be achieved for the GPU code by comparing with the CPU code running sequentially. The speed up factor is similar to the previous case, indicating that our method could handle complex geometry pretty well on GPU platform.

A weak scaling test is carried out on a sequence of four successively refined tetrahedral grids. 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 detailed timing measurements are presented in Table 10. The parallel efficiency ratios of 90.2% and 90.4% were obtained for DG(P1) and rDG(P1P2) in the case of eight GPUs, respectively. The speed up factor and the parallel efficiency shows that the presented

algorithm can handle the simulation over complex geometry pretty well. Above all, this parallel rDG solver based on the OpenACC directives exhibits a competitive scalability for computing inviscid flow problems on multiple GPUs.



(a) (b)

Figure 3: Transonic flow past a at a free-stream Mach number of $M_\infty = 0.85$ and a angle of attack of $\alpha = 2°$: (a) Unstructured mesh used for computation; (b) Computed pressure contours obtained by implicit rDG(P1P2)

Table 9: Timing measurements of using implicit rDG methods for transonic flow past a Boeing 747 aircraft.

	T_{unit} by implicit DG(P1)			T_{unit} by implicit rDG(P1P2)		
Nelem	GPU	CPU	Speedup	GPU	CPU	Speedup
64,066	11.86	90.84	7.66	15.30	102.71	6.71
127,947	11.02	90.66	8.23	13.47	101.14	7.51

Table 10: Timing measurements of weak scaling obtained on a cluster of NVIDIA Tesla C2050 GPU cards for transonic flow past a Boeing 747 aircraft.

			Unit time (ms)		Parallel efficiency	
Grids	Elements	GPU's	P1	P1P2	P1	P1P2
Level 1	64,066	×1	25.58	30.28	–	–
Level 2	127,947	×2	26.57	32.20	96.3%	94.0%
Level 3	253,577	×4	28.08	32.81	91.1%	92.3%
Level 4	512,280	×8	28.35	33.48	90.2%	90.4%

VII. Conclusion and outlook

In this study, a GPU accelerated, implicit reconstructed discontinuous Galerkin method has been developed based on the OpenACC directives for the solution of compressible inviscid flows on 3D unstructured grids. The parallelization of high-order implicit algorithm has been a historical challenge because of the nature of GPGPU. More specifically, the computation power and local cache memory size of each core on GPU are way less than those of a typical CPU core, which would require fine granularity algorithm to achieve high efficiency on GPGPU. Therefore, sub-matrix operations are refined into element level. Efforts are made to adopt a compact in-place direct inversion algorithm based on the Gauss-Jordan elimination. In addition, a special element reordering algorithm is carried out to resolve the inherent data dependency of

some featured preconditioning algorithm, like LU-SGS or SGS. Indeed, the current OpenACC directive-based algorithm may not outperform some other GPU program models like CUDA, however the biggest benefit from adopting OpenACC to a current CFD solver still stands: it only requires minimum intrusion and algorithm alteration to an existing CPU code, and renders an efficient approach to upgrade a legacy solver with the GPU-computing capability without compromising its cross-platform portability and compatibility with the mainstream compilers, which make it a remarkable design feature in the present scheme. A series of inviscid flow problems have been presented for obtaining the speed up factor as well as the parallel efficiency, indicating that the presented GPU accelerated implicit algorithm is a cost-effective method on a NVIDIA GPU card.

There is much room for further improvement of the current implicit algorithm. First of all, a fine-grained versions of the left hand side computing algorithm needs to be developed, since the memory bound issue caused by the current coarse-grained fashion has made it the major bottleneck of the current scheme. Secondly, an OpenACC directive-based GMRES scheme would need to be carried out so that the GMRES + LU-SGS scheme can be efficient on GPGPU. Finally, the work should be extended to consider viscous terms and thus Navier-Stokes equations can be solved to simulate the viscous flow problems on the GPU framework.

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. GMRES discontinuous Galerkin solution of the Compressible Navier-Stokes Equations. *Discontinuous Galerkin Methods, Theory, Computation, and Applications. Edited by B. Cockburn, G. E. Karniadakis, and C. W. Shu. Lecture Notes in Computational Science and Engineering*, 11:197–208, 2000.

[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]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.

[7]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.

[8]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.

[9]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.

[10]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.

[11]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.

[12]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.

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

[14]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.

[15]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.

[16]Rainald Löhner. *Applied computational fluid dynamics techniques: an introduction based on finite element methods*. John Wiley & Sons, 2008.

[17]J. Lou, Y. Xia, L. Luo, H. Luo, J. Edwards, and F. Mueller. OpenACC-based GPU Acceleration of a p-multigrid

Discontinuous Galerkin Method for Compressible Flows on 3D Unstructured Grids. *53rd AIAA Aerospace Sciences Meeting*, 2015-0822, 2015. http://arc.aiaa.org/doi/abs/10.2514/6.2015-0822.

[18]H. Luo, J. D. Baum, and R. Löhner. A Fast, Matrix-free Implicit Method for compressible flows on Unstructured Grids. *Journal of Computational Physics*, 146(2):664–690, 1998.

[19]H. Luo, D. Sharov, J. D Baum, and R. Löhner. Parallel unstructured grid gmres+ lu-sgs method for turbulent flows. *AIAA Paper 2003*, 273, 2003.

[20]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.

[21]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.

[22]L. Luo, J. R Edwards, H. Luo, and F. Mueller. A fine-grained block ilu scheme on regular structures for gpgpus. *Computers & Fluids*, 119:149–161, 2015.

[23]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.

[24]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.

[25]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.

[26]Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[27]D. Sharov, H. Luo, J. D Baum, and R. Löhner. Implementation of unstructured grid gmres+ lu-sgs method on shared-memory, cache-based parallel computers. *AIAA paper*, 927:2000, 2000.

[28]Dmitri Sharov and Kazuhiro Nakahashi. Reordering of hybrid unstructured grids for lower-upper symmetric Gauss-Seidel computations. *AIAA journal*, 36(3):484–486, 1998.

[29]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.

[30]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.

[31]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.

[32]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.

[33]Y. Xia, J. Lou, H. Luo, J. Edwards, and F. Mueller. Openacc acceleration of an unstructured cfd solver based on a reconstructed discontinuous galerkin method for compressible flows. *International Journal for Numerical Methods in Fluids*, 78(3):123–139, 2015.

[34]Y. Xia, J. Lou, L. Luo, H. Luo, J. Edwards, and F. Mueller. On the Multi-GPU Computing of a Reconstructed Discontinuous Galerkin Method for Compressible Flows on 3D Hybrid Grids. *7th AIAA Theoretical Fluid Mechanics Conference*, 2014-3081, 2014. http://arc.aiaa.org/doi/abs/10.2514/6.2014-3081.

[35]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.

[36]Y. Xia, H. Luo, and R. Nourgaliev. An Implicit Reconstructed Discontinuous Galerkin Method Based on Automatic Differentiation for the Navier-Stokes Equations on Tetrahedron Grids. *AIAA Paper*, 2013-0687, 2013.

[37]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.

[38]Y. Xia and R. Nourgaliev. An Implicit Hermite WENO Reconstruction-Based Discontinuous Galerkin Method on Tetrahedral Grids. *7th International Conference on Computational Fluid Dynamics*, ICCFD7-4205, 2012.

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