OpenACC-based GPU Acceleration of a 3-D Unstructured Discontinuous Galerkin Method

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

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 GPU-accelerated discontinuous Galerkin (DG) method is presented for the solution of compressible flows on 3-D unstructured grids. The present work has employed two of the most attractive features in a new programming standard of parallel computing – OpenACC: 1) multi-platform/compiler support and 2) descriptive directive interface to upgrade a legacy CFD solver with the capacity of GPU computing, without significant extra cost in recoding, resulting in a highly portable and extensible GPU-accelerated code. In addition, a face renumbering/grouping scheme is proposed to overcome the "race condition" in face-based flux calculations that occurs on GPU vectorization. Performance of the developed double-precision solver is assessed for both simple and complex geometries. Speedup factors up to but not limited to 24× and 1.6× were achieved by comparing the measured computing time of the OpenACC program running on an NVIDIA Tesla K20c GPU to that of the equivalent MPI program running on one single core and full sixteen cores of an AMD Opteron-6128 CPU respectively, indicating a great potential to port more features of the underlying DG solver into the OpenACC framework.

I. Introduction

The General-Purpose Graphics Processing Unit (GPGPU)[20] technology 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. 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 CPU-based parallel computing and the complexity of the simulation problems to be solved continues to widen. Fortunately, GPGPU offers a new opportunity to significantly accelerate CFD simulations, and is expected to be a major compute unit in the near future.

Among the available GPGPU technologies, NVIDA's CUDA application programming interface (API) and CUDA-enabled accelerators are recognized as a popular parallel programming model and platform. Thus the CUDA technology has been widely adopted in developing GPU-accelerated CFD solvers, where the numerical methods range from the finite difference methods (FDMs), spectral difference methods (SDMs) and finite volume methods (FVMs) to discontinuous Galerkin methods (DGMs). To list a few of the CUDA-accelerated CFD applications, Elsen *et al.*[11] reported a 3D high-order FDM solver for large calculation on

[*] Postdoctoral Research Associate, AIAA Member. Corresponding Author: yxia2@ncsu.edu
[†] Professor, AIAA Associate Fellow.
[‡] Postdoctoral Research Associate, AIAA Member
[§] Professor, AIAA Associate Fellow.
[¶] Graduate Student, AIAA Student Member
[‖] Professor

multi-block structured grids; Klöckner *et al.*[16] developed a 3D unstructured high-order nodal DGM solver for the Maxwell's equations; Corrigan *et al.*[10] proposed a 3D FVM solver for compressible inviscid flows on unstructured tetrahedral grids; Zimmerman *et al.*[29] presented an SDM solver for the Navier-Stokes equations on unstructured hexahedral grids; and more as in the references.[3,4,12,7,21,23,13,19,8,14,1,9]

However applying CUDA to a legacy CFD code is not likely 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.[15] So what if the CFD code designers have to meet specific investment requirements like (1) enable GPU computing for legacy CFD programs at a minimum extra cost in time and effort (usually a major concern for large-scale code development), (2) enable the GPU-accelerated programs running on different platforms (similar to the situation that the video game designers would like to make their products available across platforms)? Moreover, designers may also have concerns like what if they cannot afford the time to adopt to a new programming language like CUDA and rewrite the source code, and what if they do not plan to depend on a proprietary language like CUDA for long and tie their applications to the CUDA-enabled devices only? Provided with the fact that many legacy and current CFD programs (including the one presented) are intended to maintain versatility and portability over multiple platforms even with the capacity of GPU computing, thus adopting to CUDA might spell almost a brand new design and long-term project, and a constraint to the CUDA-enabled devices. Fortunately, CUDA is not the sole player in this scope. Two other solutions include OpenCL[22] — the currently dominant open GPGPU computing language (but disregarded from further discussion since it does not support Fortran), and OpenACC[24] — a new open parallel programming standard which is designed in order to meet the two requirements as indicated above.

Much like in OpenMP, developers working with OpenACC simply needs to annotate their source code to identify the areas that should be accelerated using the compiler directives and some additional functions, without much effort to modify the source code as to accommodate to specific GPU architectures. Thus developers will benefit not only from easy implementation of the directives but also the freedom to run the very same accelerated code on either CPU or GPU of different vendors, e.g., NVIDIA, AMD and Intel accelerators. However OpenACC may still lag behind CUDA for some cutting-edge features due to the vendor's distribution plan (note that NVIDIA is among OpenACC's supporter organizations), But in compensation OpenACC presents a design style for developing a unified codebase that is promised to provide multi-platform and multi-vendor compatibility, offering an ideal way to minimize investment in legacy programs by enabling an easy migration path to accelerated computing.

The objective of the effort in the present work is to develop a GPU-accelerated discontinuous Galerkin method for the solution of compressible flows on 3-D unstructured grids. Two of the most attractive features in OpenACC: 1) multi-platform/compiler support and 2) easy-to-code directives has been employed to partially upgrade a legacy Navier-Stokes DG solver[17,18,26,27,25,28] with the GPU computing capacity without significant extra cost in recoding and maintenance, resuling in a highly portable and extensible GPU codebase. A face renumbering/grouping scheme is proposed in order to solve the "race condition" in face-based flux calculations that occurs on GPU vectorization. Performance of the developed double-precision solver is assessed for both simple and complex geometries. Speedup factors up to but not limited to $24\times$ and $1.6\times$ were achieved by comparing the measured computing time of the OpenACC program running on an NVIDIA Tesla K20c GPU to that of the equivalent MPI program running on one single core and full sixteen cores of an AMD Opteron-6128 CPU respectively, indicating a great potential to port more features of the underlying DG solver into the OpenACC framework.

The outline of the rest of the paper is organized as follows. In Section 1, the governing equations are briefly introduced. In Section 2, the discontinuous Galerkin spatial discretization is described. In Section 3, the keynotes of porting an unstructured DGM solver to GPU with the OpenACC directives is illustrated. In Section 4, the results of scaling test cases are presented and analyzed. The concluding remarks and plan of future work are given in Section 5.

II. Governing Equations

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

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

where the summation convention has been used. The conservative variable vector $\mathbf{U}$, 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

$$p = (\gamma - 1)\rho\left(e - \frac{1}{2}(u^2 + v^2 + w^2)\right) \tag{3}$$

which is valid for perfect gas. The ratio of the specific heats γ is assumed to be constant and equal to 1.4.

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

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

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,

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

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. The flux function $\mathbf{F}_k(\mathbf{U}_h)\mathbf{n}_k$ appearing in the second terms of Eq. (6) is replaced by a numerical Riemann flux function $\mathbf{H}_k(\mathbf{U}_h^L, \mathbf{U}_h^R, \mathbf{n}_k)$ where $\mathbf{U}_h^L$ and $\mathbf{U}_h^R$ are the conservative state vectors at the left and right side of the element boundary. 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. The inviscid flux is evaluated by the HLLC[2] scheme. To move the second and third terms to the right-hand-side in Eq. 6, it leads to a system of ordinary differential equations (ODEs) in time and Eq. 6 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 a hierarchical WENO reconstructed scheme to improve the accuracy and non-linear stability of the underlying second-order linear polynomial DG(P1) solution, as the details are referred in the authors' prior works.[17,18]

IV. OpenACC Implementation

The computation-intensive portion of the underlying unstructured discontinuous Galerkin solver is a time marching loop which repeatedly computes the time derivatives of the conservative variable vector,

as shown in Eq. 7. The conservative variable vector is updated using the three-stage TVD Runge-Kutta time stepping scheme[5,6] within each time loop. The most expensive workload when computing the time derivatives consists of two procedures: accumulating the right hand side of Eq. 7 from the contribution of integral over interior/boundary faces, and the contribution of integral over the elements. In fact, the way to add the OpenACC directives in a legacy code is quite easy and similar to OpenMP, as the example shown in Table 1 which demonstrates the parallelization of a loop over the elements for collecting contribution to the elemental-stored 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 integration points over an element. Both the OpenMP and OpenACC parallel construct directives can be applied to a readily vectorizable loop like in Table 1 without the need to change the original code.

Table 1: A vectorizable example of OpenMP / OpenACC parallel construct derivatives implementation.

```
!... OpenMP version:
!... loop over the elements

!$omp parallel
!$omp do
do ie = 1, Nelem
  do ig = 1, Ngp
    !... contribution to the element
    rhsel(*,*,ie)=rhsel(*,*,ie)+flux
  enddo
enddo
!$omp end parallel
```

```
!... OpenACC version
!... loop over the elements

!$acc parallel
!$acc loop
do ie = 1, Nelem
  do ig = 1, Ngp
    !... contribution to the element
    rhsel(*,*,ie)=rhsel(*,*,ie)+flux
  enddo
enddo
!$acc end parallel
```

However due to the unstructured grid topology, the attempt to directly wrap a loop over the faces for collecting contribution to the residual vector with either the OpenMP or OpenACC directives can lead to the so-called "race condition" as the example is shown in Fig. 2, where multiple writes to the same residual vector will occur, and thus can smear the correct values in vectorized computing. One feasible solution

Table 2: A "race condition" example of OpenMP / OpenACC parallel construct derivatives implementation.

```
!... OpenMP version (race condition):
!... loop over the faces

!$omp parallel
!$omp do
do ifa = Njfac+1, Nafac
  iel = intfac(1,ifa) ! left  element
  ier = intfac(2,ifa) ! right element
  do ig = 1, Ngp
    !... contribution to the left
    rhsel(*,*,iel)=rhsel(*,*,iel)-flux
    !... contribution to the right
    rhsel(*,*,ier)=rhsel(*,*,ier)+flux
  enddo
enddo
!$omp end parallel
```

```
!... OpenACC version (race condition):
!... loop over the faces

!$acc parallel
!$acc do
do ifa = Njfac+1, Nafac
  iel = intfac(1,ifa) ! left  element
  ier = intfac(2,ifa) ! right element
  do ig = 1, Ngp
    !... contribution to the left
    rhsel(*,*,iel)=rhsel(*,*,iel)-flux
    !... contribution to the right
    rhsel(*,*,ier)=rhsel(*,*,ier)+flux
  enddo
enddo
!$acc end parallel
```

to the issue of "race condition" is changing the code structure so as to collect the face contributions in the loop over elements as proposed for the finite volume methods in Ref.,[10] but at the price of redundant computation. According to Ref.,[10] the performance of this approach implemented for the FVM solver was apparently advantageous only in single-precision GPU computation. It also needs to be noted that the face loops of DGMs differ mainly from those of FVMs with an extra inner loop over the integration points, where Ngp denotes the number of Gauss quadrature points. Ngp is equal to 1 for the FVMs, but can be quite a larger number for the DGMs, and thus would cause much higher overhead if such computation is doubled. Furthermore, this approach also requires a change of the major loop structure and construction of a new element-face connectivity matrix, making the equivalent CPU code not efficient at all. Alternatively, the "race condition" can be eliminated with only a few modifications of the face loop structures, by adopting to a face renumbering scheme. The scheme is designed in order to divide the faces into several groups, ensuring that no two faces that belong to a common element fall in the same group. Consequently, an additional

do-construct that loops over these groups is nested on top of the original face loop and executed sequentially, as shown in Table 3. Therefore the inner do-construct that loops over the faces is vectorized without the "race condition" issue. The number of the groups is usually 6, 7 or 8 according to a wide range of test cases, and can be set equal to 1 for the equivalent CPU code, which recovers to exactly the original code. This face renumbering scheme results in a unified CPU/GPU unstructured code across platforms. In addition, it needs to be mentioned that a more direct solution to this issue might be applying the atomic construct (available since OpenACC specification Version 2.0) to prevent simultaneous reading / writing by vector threads even without the presented renumbering scheme. However, the compiler version used in this work is not up-to-date enough to support this functionality. Thus the atomic construct will be tested in future once the compiler implements it.

Table 3: A no-race-condition example of OpenMP / OpenACC parallel construct derivatives implementation.

```
!... OpenMP version (no race condition):
!... loop over each group of faces

Nfac1 = Njfac
do ipass = 1, Npass_ift
  Nfac0 = Nfac1 + 1
  Nfac1 = fpass_ift(ipass)
  !$omp parallel
  !$omp do
  do ifa = Nfac0, Nfac1
    iel = intfac(1,ifa) ! left  element
    ier = intfac(2,ifa) ! right element
    do ig = 1, Ngp
      !... contribution tothe left
      rhsel(*,*,iel)=rhsel(*,*,iel)-flux
      !... contribution to the right
      rhsel(*,*,ier)=rhsel(*,*,ier)+flux
    enddo
  enddo
  !$omp end parallel
enddo
```

```
!... OpenACC version (no race condition):
!... loop over each group of faces
!
Nfac1 = Njfac
do ipass = 1, Npass_ift
  Nfac0 = Nfac1 + 1
  Nfac1 = fpass_ift(ipass)
  !$acc parallel
  !$acc do
  do ifa = Nfac0, Nfac1
    iel = intfac(1,ifa) ! left  element
    ier = intfac(2,ifa) ! right element
    do ig = 1, Ngp
      !... contribution to the left
      rhsel(*,*,iel)=rhsel(*,*,iel)-flux
      !... contribution to the right
      rhsel(*,*,ier)=rhsel(*,*,ier)+flux
    enddo
  enddo
  !$acc end parallel
enddo
```

Since the main time marching loop is performed on GPU, a one-time data copy between the host CPU and the attached GPU can be done at the beginning and the end of the complete solving with minimal overhead, as shown in Table 4. This data translation accounts for a very small portion of overhead throughout the solving process, which might take the solver about seconds to minutes, depending on the scale of the computational grids.

Table 4: An example of the main time marching loop.

```
!... OpenMP version: time marching loop

!... No data translation API directive
!... is needed since
!... all the data is on CPU memory
do itime = ninit, ntime
  !... explicit time stepping scheme
  call exrkdg3d()
enddo
```

```
!... OpenACC version: time marching loop

!$acc data region &
!$acc copy(unkno,unold,rhsel) &
!$acc copyin(...other arrays...)
do itime = ninit, ntime
  !... explicit time stepping scheme
  call exrkdg3d()
enddo
!$acc end data region
```

The present discontinuous Galerkin solver code is written in Fortran 95 and compiled with the PGI Accelerator compiler suite. If a CUDA-enabled GPU hardware is linked, the PGI compiler will first translate the subroutines that contain OpenACC parallel loops into the CUDA language, and then generate CUDA acceleration kernels. Therefore in this case, the runtime performance of the resulting OpenACC program is determined partially by the indirectly generated CUDA kernels that implement the actual computation on GPU. On the other side, although the OpenACC standards are almost complete (up to version 2.0 as of now), more time is expected for its supporting vendors like PGI and CAPS to release products that target a wider range of graphics hardware, e.g., AMD line of accelerated processing units (APUs) as well as the AMD line of discrete GPU accelerators.

V. Numerical Results

The performance of the OpenACC-based GPU program was measured on a NVIDIA Tesla K20c GPU containing 2496 multiprocessors. The performance of the equivalent MPI-based parallel CPU program was measured on an AMD Opteron 6128 CPU containing 16 cores. The source codes were compiled with the PGI Accelerator (version 13.4) + OpenMPI (version 1.5.5) compiler/runtime suite. A few quantities for timing measurements are defined. Firstly, the unit running time T_{unit} is calculated as

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

where the running time T_{run} refers to the time recorded only for completing the entire time marching loop with a given number of time steps Ntime, not including the start-up procedures, initial/end data translation, and solution file dumping. Secondly, the speedup factors were obtained by comparing the timings of the GPU program against those obtained with the CPU program running on one and sixteen cores (denoted as CPU-1 and CPU-16), respectively. However the portability feature of the OpenACC code is not demonstrated in this work due to the limited compiler/accelerator resources.

V.A. Simple Geometry: Subsonic Flow past a Sphere

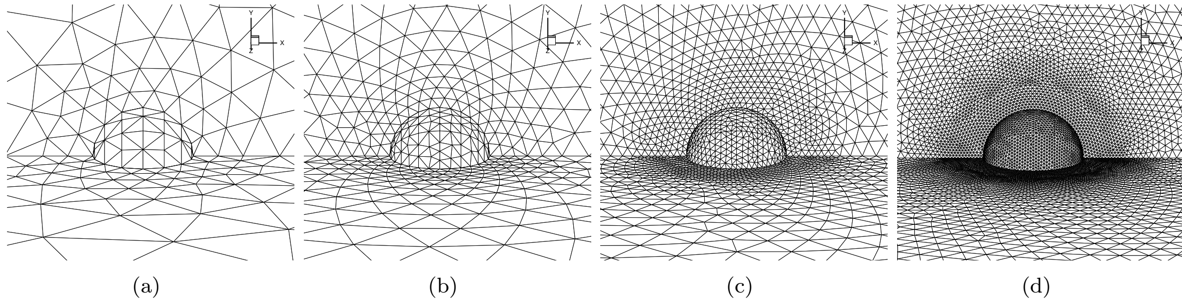
The test case of an inviscid subsonic flow past a sphere at a free stream Mach number of $M_\infty = 0.5$ is considered first in order to verify and validate the OpenACC code, and assess the OpenACC acceleration on different levels of computing scales. To serve these purposes, a sequence of four successively refined tetrahedral grids are used in this computation, as shown in Figs. 1(a) – 1(b) respectively: the Level-1 grid consists of 2,426 elements, 589 nodes and 640 boundary faces; the Level-2 grid consists of 16,467 elements, 3,425 nodes and 2,372 boundary faces; the Level-3 grid consists of 124,706 elements, 23,462 nodes and 9,072 boundary faces; and the Level-4 grid consists of 966,497 elements, 172,512 nodes and 34,662 boundary faces. The cell size is halved between consecutive grids. Note that only a quarter of the configuration is modeled due to symmetry of the problem. The computation is started with a uniform flow field and the number of iteration for each grid is large enough for the flow field to converge to the steady state, as illustrated by the surface pressure contours in Figs. 2(a) – 2(b). The timing measurements are given in Table 5, along with Figs. 3 and 4 demonstrating the unit running time and speedup factors, respectively. From these results, one can see that GPU acceleration for small-scale computing like on the Level-1 grid may not gain advantage over the 16-core CPU compute node. With the increased computing scale like on the Level-2, 3 and 4 grids, the potential of OpenACC is tapped, as one can observe a series of speedup factors up to 22.6× and 1.49× by comparing to the CPU program running on one single core and on full sixteen cores of the CPU compute node, respectively. Moreover, it can be seen in Fig. 4 that a further improving speedup factor is to be expected provided with a larger computing scale.

Table 5: Timing measurements for subsonic flow past a sphere.

	T_{unit} (microsecond)			Speedup	
Nelem	GPU	CPU-1	CPU-16	vs. CPU-1	vs. CPU-16
2,426	20.2	176.8	14.8	8.8	0.73
16,467	10.7	182.8	12.6	17.0	1.18
124,706	9.3	182.8	13.0	19.6	1.40
966,497	8.8	198.9	13.1	22.6	1.49

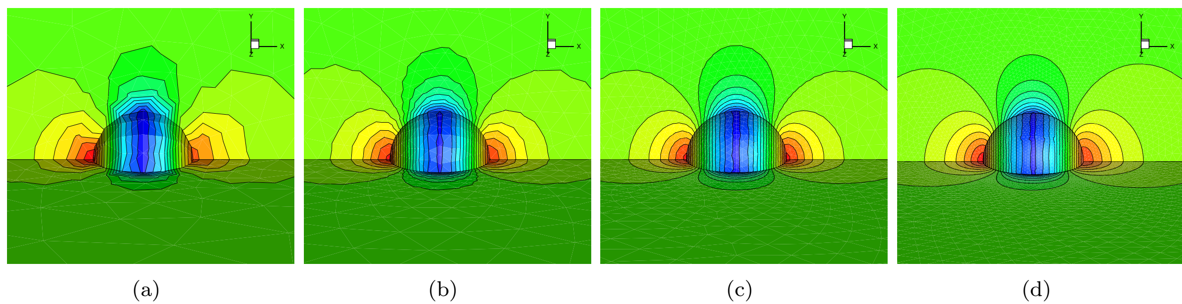
V.B. Complex Geometry: Transonic Flow over a Boeing 747 Aircraft

The test case of a transonic flow past a complete Boeing 747 aircraft at a free stream Mach number of $M_\infty = 0.85$ and an angle of attack of $\alpha = 2°$ is chosen in order to assess the performance of the OpenACC-based GPU program in computing complex geometric configurations. The Boeing 747 configuration includes the fuselage, wing, horizontal and vertical tails, under-wing pylons, and flow-through engine nacelle. Two



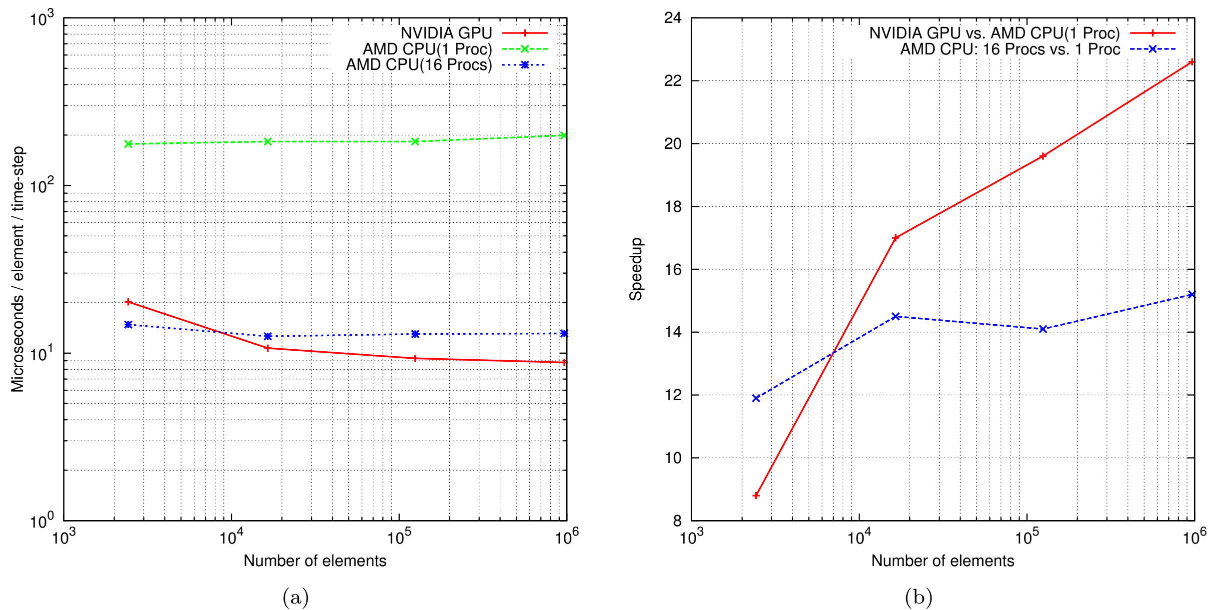
(a) (b) (c) (d)

Figure 1: The four successively refined tetrahedral grids for subsonic flow past a sphere.



(a) (b) (c) (d)

Figure 2: Surface pressure contours obtained on the four successively refined tetrahedral grids for subsonic flow past a sphere.



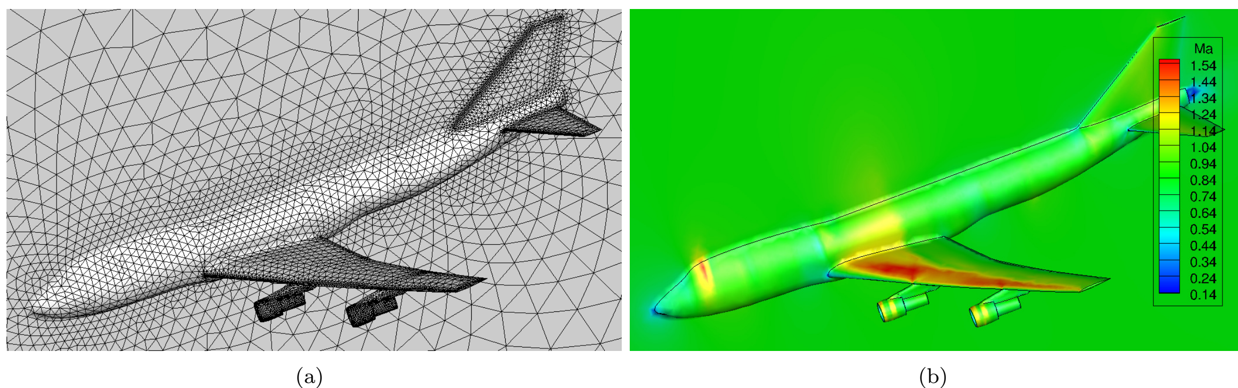
(a) (b)

Figure 3: Statistics for subsonic flow past a sphere in double precision: (a) unit running time; (b) performance scaling.

grids are used in this computation, containing 253,577 elements, 48,851 grid points, 23,616 boundary faces, and 1,025,170 elements, 371,162 grid points, 60,780 boundary faces, respectively. Note that only the half-span airplane is modeled due to symmetry of the problem, as shown in Fig. 5. The computation is started with a uniform flow field and iterated until the steady state is reached, as illustrated by the surface Mach number contours also in Fig. 6. Timing measurements are given in Table 6, along with Figs. 7 and 8 demonstrating the unit running time and speedup factors, respectively. From the results, one can observe two sets of speedup factors up to 24.5× and 1.57× by comparing to the CPU program running on one single core and on full sixteen cores of the CPU compute node, respectively. Overall, the highest speedup factors observed are similar to the first test case.

Table 6: Timing measurements for transonic flow over a Boeing 747 Aircraft.

	T_{unit} (microsecond)			Speedup	
Nelem	GPU	CPU-1	CPU-16	vs. CPU-1	vs. CPU-16
253,577	11.5	243.8	16.4	21.2	1.43
1,025,170	10.6	249.4	16.6	24.5	1.57



(a) (b)

Figure 4: Illustration of transonic flow over a Boeing 747 aircraft at $M_\infty = 0.85$ and $\alpha = 2°$: (a) surface unstructured triangular meshes; (b) surface Mach number contours.

VI. Conclusion and Outlook

A new GPU parallel programming standard — OpenACC, has been explored in this paper to port a legacy unstructured discontinuous Galerkin flow solver to the GPU accelerator without significant extra effort in adjusting the original code. A face renumbering/grouping scheme is proposed in order to overcome the "race condition" in face-based flux calculations that occurs on GPU vectorization. The scaling test conducted in this paper has demonstrated good speedup for the resulting GPU-accelerated code on both simple and complex geometries, indicating the potential to further port more features of the underlying solver onto GPU parallel computing with OpenACC.

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.

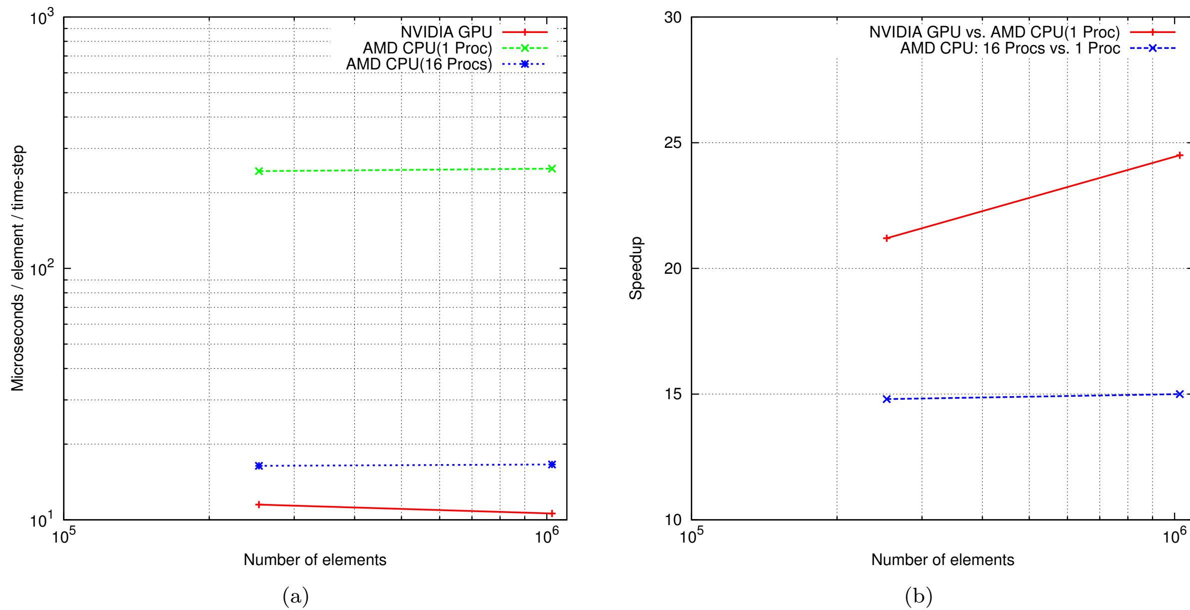


Figure 5: Statistics for transonic flow over a Boeing 747 aircraft in double-precision: (a) unit running time; (b) performance scaling.

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]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.

[3]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.

[4]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.

[5]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.

[6]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.

[7]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.

[8]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.

[9]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.

[10]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.

[11]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.

[12]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.

[13]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.

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

[15]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.

[16]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.

[17]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.

[18]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.

[19]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.

[20]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.

[21]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.

[22]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.

[23]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.

[24]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.

[25]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.

[26]Y. Xia, H. Luo, and R. Nourgaliev. An Implicit Method for a Reconstructed Discontinuous Galerkin Method on Tetrahedron Grids. *AIAA Paper*, 2012-2834, 2012.

[27]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.

[28]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*, 2013-3062, 2013.

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