Development of a Portable, GPU-Accelerated High-Order Discontinuous Galerkin CFD Code for Compressible Flows on Hybrid Grids

Yidong Xia, Lixiang Luo, Jialin Lou, Hong Luo and Jack Edwards
Department of Mechanical and Aerospace Engineering
North Carolina State University

and

Frank Mueller and Nishanth Balasubramanian
Department of Computer Science
North Carolina State University

February 7, 2014
# People at NC State

## Faculty (Co-PI’s)

<table>
<thead>
<tr>
<th>Name</th>
<th>Title and Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Hong Luo</td>
<td>Professor, Department of Mechanical and Aerospace Engineering</td>
</tr>
<tr>
<td>Dr. Jack Edward</td>
<td>Professor, Department of Mechanical and Aerospace Engineering</td>
</tr>
<tr>
<td>Dr. Frank Mueller</td>
<td>Professor, Department of Computer Science</td>
</tr>
</tbody>
</table>

## Postdocs

<table>
<thead>
<tr>
<th>Name</th>
<th>Role and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Yidong (Tim) Xia</td>
<td>Current principal developer of the RDGFLO code</td>
</tr>
<tr>
<td>Dr. Lixiang (Eric) Luo</td>
<td>Investigation of frontier challenges and solutions in GPU computing; developer of INCOMP3D</td>
</tr>
</tbody>
</table>

## Students

<table>
<thead>
<tr>
<th>Name</th>
<th>Title and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mr. Jialin (Johnny) Lou</td>
<td>Ph.D. student of Aerospace Engineering; current developer of the RDGFLO code</td>
</tr>
<tr>
<td>Mr. Nishanth Balasubramanian</td>
<td>Master student of Computer Science; investigation of GPU computing models for CFD programming</td>
</tr>
</tbody>
</table>
Outline

① A brief overview
② Motivation and objective
③ Governing equations of fluid dynamics
④ Discontinuous Galerkin formulation
⑤ OpenACC-based parallelism
⑥ Numerical examples
⑦ Concluding remarks
⑧ Future work
# GPU Computing in CFD – Overview

## GPGPU

**General-purpose computing on graphics processing units**

- The utilization of GPU, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by CPU.

## Why?

The great potential and scalability of GPGPU for CFD applications!

## How?

Offload the computing-intensive portion from the host CPUs to the GPU devices.

## Programming models

- **OpenCL**: the currently dominant open GPGPU programming language
- **NVIDIA’s CUDA**: the dominant proprietary framework
- **OpenACC**: a collection of directives designed to simplify parallel programming
GPU Computing in CFD – Overview

A variety of applications (selected)

• For the finite difference method (FDM)

• For the finite volume method (FVM)

• For the spectral difference method (SDM)

• For the discontinuous Galerkin method (DGM)

Observation: most GPU-related CFD solvers are based on CUDA
Motivation & Objectives

Motivation

Tap the power of GPU parallel computing for the aerodynamic design of Unmanned Aerial Vehicle (UAV) with CFD toolkits

MQ-1 Predator UAV (Source: grabcad.com CAD by Chao Wey)
# Motivation & Objectives

## Objectives

A portable, efficient and ultimately competitive GPU parallelization strategy for extensible and sustainable high-fidelity CFD code development

## An unavoidable debate – CUDA, OpenCL or OpenACC?

It really depends on your needs, e.g., for us, **cross-platform portability**
- Support from a wide range of compiler and accelerator vendors
- C/C++ and **Fortran**
- Best performance
- Less fine-tuning effort, especially for a legacy code package
- Available computing resource

[Logos for CUDA, OpenCL, and OpenACC]
GPU-Computing Framework

Option 1. – based on NVIDIA’s CUDA

• Why CUDA?
  - Mature and ever-updating GPU parallelization standards for HPC
  - Currently wide user community support
  - Achievable optimal performance with fine-tuning
  - Strong GPU-accelerated library support, e.g., CULA tools
  - ……

• Why not CUDA?
  - Complex and explicit layout of threads on GPU for each kernel function
  - Excessive workload to upgrade an existing CFD package
  - Uncertainty in the vendor’s long-term development strategy
  - Constrained portability of the developed code on non-CUDA devices

CUDA alone can not satisfy not only our primary design goals, but also many others who hesitate to adopt GPU computing!
**GPU-Computing Framework**

## Option 2. – based on OpenACC

### Directives for accelerators
http://www.openacc-standard.org/

- **Why OpenACC?**
  - Simple directive-based GPU parallelization strategy, similar to OpenMP
  - Multi-compiler / multi-platform support
  - Growing supporting community

- **Why not OpenACC?**
  - If a fine-tuned, best-performance code is what you pursue
  - A number of limitations compared with CUDA

---

**Our choice: OpenACC meets most of our design requirements!**
### Desirable features of the resulting CFD code

- **Multi-compiler compatibility**
  - GNU Fortran compiler
  - Intel Fortran compiler
  - PGI Accelerator Fortran compiler (with OpenACC support)
  - CAPS Fortran compiler (with OpenACC support)

- **Cross-platform portability**
  - Intel CPUs
  - AMD CPUs / APUs (with potential OpenACC support in 2014)
  - NVIDIA CUDA-enabled GPUs (with OpenACC support)

- **Extensible and sustainable programming schemes**
- **Competitive performance**
Legacy CFD Package

RDGFLO – a baseline code for OpenACC-based GPU parallelization

- **Reconstructed Discontinuous Galerkin finite element FLOw solver**
  - High-order solution of compressible flows on 3-D hybrid grids
  - Explicit / implicit solution schemes
  - Domain-partition based MPI parallel computing

Gallery

ONERA M6 wing  wing/pylon/finned-store configuration  Boeing 747 aircraft
The Navier-Stokes equations for unsteady compressible flows

\[
\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_k}{\partial x_k} = \frac{\partial \mathbf{G}_k}{\partial x_k}
\]

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

\[
\mathbf{U} = \begin{pmatrix} \rho \\ \rho u_i \\ \rho e \end{pmatrix}, \quad \mathbf{F}_j = \begin{pmatrix} \rho u_j \\ \rho u_i u_j + p \delta_{ij} \\ u_j (\rho e + p) \end{pmatrix}, \quad \mathbf{G}_j = \begin{pmatrix} 0 \\ \tau_{ij} \\ u_i \tau_{ij} + q_j \end{pmatrix}
\]
The Navier-Stokes equations for unsteady compressible flows (cont.)

The pressure $p$ can be computed from the equation of state (EOS)

$$p = (\gamma - 1)(\rho e - \frac{1}{2} \rho u_k u_k)$$

which is valid for perfect gas. The ratio of 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

$$\tau_{ij} = \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij}$$

$$q_j = \frac{1}{\gamma - 1} \frac{\mu}{Pr} \frac{\partial T}{\partial x_j}$$

where $T$ is the temperature of the fluid, $Pr$ the laminar Prandtl number, which is 0.7 for air. $\mu$ represents the molecular viscosity, which can be determined through the Sutherland’s law

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

where $\mu_0$ is the viscosity at the reference temperature $T_0$ and $S = 110K$. 
Discontinuous Galerkin Method

Weak formulation of the governing equations

\[
\frac{d}{dt} \int_{\Omega_e} U_h B_i \, d\Omega + \int_{\Gamma_e} F_k n_k B_i \, d\Gamma - \int_{\Omega_e} F_k \frac{\partial B_i}{\partial x_k} \, d\Omega =
\]
\[
\int_{\Gamma_e} G_k n_k B_i \, d\Gamma - \int_{\Omega_e} G_k \frac{\partial B_i}{\partial x_k} \, d\Omega, \ 1 \leq i \leq N
\]

where \( B_i(x) \) is the basis function of polynomials of degree \( p \), and \( N \) is the dimension of the polynomial space \( p \).

The Taylor-basis discontinuous Galerkin (DG\((p)\)) solution in each element

\[
U = \sum_{i=1}^{N} U_i B_i
\]

For example, the underlying piecewise linear polynomial DG(P1) solution

\[
U = \bar{U} + \left( \frac{\partial U}{\partial x} \right) \frac{\Delta x}{\Delta x} + \left( \frac{\partial U}{\partial y} \right) \frac{\Delta y}{\Delta y} + \left( \frac{\partial U}{\partial z} \right) \frac{\Delta z}{\Delta z}
\]

where \((x_c, y_c, z_c)\) is the coordinate of the element center.
Discontinuous Galerkin Method

### Hierarchical WENO reconstruction

<table>
<thead>
<tr>
<th><strong>A quadratic polynomial DG(P2) solution is obtained via a hierarchical WENO reconstruction approach in each element</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Least-squares reconstruction to obtain an initial quadratic solution</td>
</tr>
<tr>
<td>2) WENO reconstruction for the second derivatives to maintain linear stability</td>
</tr>
<tr>
<td>3) WENO reconstruction for the first derivatives to maintain nonlinear stability</td>
</tr>
</tbody>
</table>

**Reference articles**


Discontinuous Galerkin Method

Semi-discrete form

A system of ordinary differential equations (ODEs) in time

\[ M \frac{dU}{dt} = R \]

where \( M \) is the mass matrix and \( R \) is the residual vector.

Three-stage TVD Runge-Kutta (TVDRK3) time stepping

\[ U^{(1)} = U^n + \Delta tM^{-1}R(U^n) \]
\[ U^{(2)} = \frac{3}{4} U^n + \frac{3}{4} \left( U^{(1)} + \Delta tM^{-1}R(U^{(1)}) \right) \]
\[ U^{n+1} = \frac{1}{3} U^n + \frac{2}{3} \left( U^{(2)} + \Delta tM^{-1}R(U^{(2)}) \right) \]
Design of OpenACC Parallel Regions

Computing-intensive regions

1. ~65% Internal & boundary face integral: loop over mesh faces $T_e$

$$\int_{\Gamma_e} F_k n_k B_i d\Gamma$$
$$\int_{\Gamma_e} G_k n_k B_i d\Gamma$$

2. ~25% Domain integral: loop over mesh elements $\Omega_e$

$$\int_{\Omega_e} F_k \frac{\partial B_i}{\partial x_k} d\Omega$$
$$\int_{\Omega_e} G_k \frac{\partial B_i}{\partial x_k} d\Omega$$

3. ~5% TVDRK3 time stepping: loop over mesh elements $\Omega_e$
Design of OpenACC Parallel Regions

Example: a readily vectorizable region

Domain integral: loop over mesh elements \( \Omega_c \)

\[
\int_{\Omega_c} F_k \frac{\partial B}{\partial x_k} d\Omega
\]

<table>
<thead>
<tr>
<th>OpenMP version</th>
<th>OpenACC version</th>
</tr>
</thead>
<tbody>
<tr>
<td>$omp parallel $omp do do ie = 1, Nelem do ig = 1, Ngp !... contribution to this element ( \text{rhsel}(::, ie) = \text{rhsel}(::, ie) + \text{flux} ) enddo enddo $omp end parallel</td>
<td>$acc parallel $acc loop do ie = 1, Nelem do ig = 1, Ngp !... contribution to this element ( \text{rhsel}(::, ie) = \text{rhsel}(::, ie) + \text{flux} ) enddo enddo $acc end parallel</td>
</tr>
</tbody>
</table>
**Design of OpenACC Parallel Regions**

Example: a region that is **not** readily vectorizable

Face integral: loop over mesh faces \( T_e \int_{\Gamma_e} F_k n_k B_i d\Gamma \)

<table>
<thead>
<tr>
<th>OpenMP version</th>
<th>OpenACC version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!$omp parallel</code></td>
<td><code>!$acc parallel</code></td>
</tr>
<tr>
<td><code>!$omp do</code></td>
<td><code>!$acc do</code></td>
</tr>
<tr>
<td>do ifa = Njfac+1, Nafac</td>
<td>do ifa = Njfac+1, Nafac</td>
</tr>
<tr>
<td>do ig = 1, Ngp</td>
<td>do ig = 1, Ngp</td>
</tr>
<tr>
<td><img src="highlight" alt="... contribution to the face-left element" /></td>
<td><img src="highlight" alt="... contribution to the face-left element" /></td>
</tr>
<tr>
<td><code>rhsel(:, :, iel) = rhsel(:, :, iel) - flux</code></td>
<td><code>rhsel(:, :, iel) = rhsel(:, :, iel) - flux</code></td>
</tr>
<tr>
<td><img src="highlight" alt="... contribution to the face-right element" /></td>
<td><img src="highlight" alt="... contribution to the face-right element" /></td>
</tr>
<tr>
<td><code>rhsel(:, :, ier) = rhsel(:, :, ier) + flux</code></td>
<td><code>rhsel(:, :, ier) = rhsel(:, :, ier) + flux</code></td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>!$omp end parallel</td>
<td>!$acc end parallel</td>
</tr>
</tbody>
</table>

“race condition” – multiple writes to the same elemental residual vector!
Design of OpenACC Parallel Regions

A common method for avoiding “race condition” in face integral

- Perform face integral at the element level
  - All the computing is implemented as loops over mesh elements
- Overheads
  - Lead to redundant computation of face integrals (doubled!)
  - Require an additional element-face connectivity array
- Reference articles on unstructured DG/FV methods (with CUDA)

This design approach requires a major rebuild in code structures!
Design of OpenACC Parallel Regions

An alternative approach to avoiding “race condition” in face integral

- Face renumbering (coloring method)
  - renumber the faces and split them into several groups.
  - Any two faces that share a common cell do not reside in the same group.
  - The original face loops are nested in a sequential loop over groups.

An example of “face renumbering & grouping”

Before

<table>
<thead>
<tr>
<th>Physical boundary faces</th>
<th>Partition boundary faces</th>
<th>Internal faces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Physical boundary faces</th>
<th>Partition boundary faces</th>
<th>Internal faces</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 groups</th>
<th>3 groups</th>
<th>6 groups</th>
</tr>
</thead>
</table>
Design of OpenACC Parallel Regions

Example: face loops after “face renumbering & grouping”

The original face loops are nested in a sequential loop over groups.

<table>
<thead>
<tr>
<th>OpenMP version</th>
<th>OpenACC version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nfac1 = Njfac</td>
<td>Nfac1 = Njfac</td>
</tr>
<tr>
<td>do ipass = 1, Npass_ift</td>
<td>do ipass = 1, Npass_ift</td>
</tr>
<tr>
<td>Nfac0 = Nfac1 + 1</td>
<td>Nfac0 = Nfac1 + 1</td>
</tr>
<tr>
<td>Nfac1 = fpass_ift(ipass)</td>
<td>Nfac1 = fpass_ift(ipass)</td>
</tr>
<tr>
<td>!$omp parallel</td>
<td>!$acc parallel</td>
</tr>
<tr>
<td>!$omp do</td>
<td>!$acc do</td>
</tr>
<tr>
<td>do ifa = Nfac0, Nfac1</td>
<td>do ifa = Nfac0, Nfac1</td>
</tr>
<tr>
<td>do ig = 1, Ngp</td>
<td>do ig = 1, Ngp</td>
</tr>
<tr>
<td>!$... contribution to the face-left element</td>
<td>!$... contribution to the face-left element</td>
</tr>
<tr>
<td>rhsel(:, ::, iel) = rhsel(:, ::, iel) - flux</td>
<td>rhsel(:, ::, iel) = rhsel(:, ::, iel) - flux</td>
</tr>
<tr>
<td>!$... contribution to the face-right element</td>
<td>!$... contribution to the face-right element</td>
</tr>
<tr>
<td>rhsel(:, ::, iel) = rhsel(:, ::, iel) + flux</td>
<td>rhsel(:, ::, iel) = rhsel(:, ::, iel) + flux</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>!$omp end parallel</td>
<td>!$acc end parallel</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
</tbody>
</table>
Design of OpenACC Parallel Regions

“Face renumbering & grouping” vs. “element loops”

• Advantages
  - No redundant computation.
  - Least intrusion to the original code structures.
  - Recoverable to CPU parallel computing.

• Disadvantages
  - Sequential groups lead to overheads in initializing more parallel kernels.

• Remarks
  1. No direct comparison of these two strategies is yet conducted for our solver. It is unknown which will render better performance on GPU.
  2. Since *portability* is our design priority, the current approach is a preferred, although *optimal performance* might be compromised.
Hardware/Software Resource

CPU
- 2-way SMPs with AMD Opteron 6128 (Many Core) with 8 cores per socket (16 cores per node)
  - 32 GB DRAM
  - 2.0 GHz core-speed for 6128 Opteron (single core)

GPU
- NVIDIA Tesla K20c
  - Memory amount: 5.0 GB
  - Stream processors: 2496

Software (64 bit)
- Operating system
  - CentOS 5.7 Linux x86 64
- Compilation & runtime suite
  - PGI Accelerator Fortran Compiler (ver. 13.4)
  - OpenMPI (ver. 1.5.5)
Performance Assessment

Timing measurements

Unit running time $T_{\text{unit}}$

$$T_{\text{unit}} = \frac{T_{\text{run}}}{N_{\text{time}} \times N_{\text{elem}}} \times 10^6 \text{ (microsecond)}$$

where the running time $T_{\text{run}}$ refers to the time recorded for completing the time marching loop with a given iteration number $N_{\text{time}}$.

Control sets (simulations in double-precision arithmetic)

<table>
<thead>
<tr>
<th>Control Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU-1</td>
<td>Parallel computing with <strong>1 GPU device</strong></td>
</tr>
<tr>
<td>CPU-1</td>
<td>Serial computing with <strong>1 CPU core</strong></td>
</tr>
<tr>
<td>CPU-16</td>
<td>Parallel computing with <strong>16 CPU cores</strong> (domain partitioning + MPI)</td>
</tr>
</tbody>
</table>
Example 1. Subsonic Flow past a Sphere

Subsonic flow past a sphere at $M_\infty = 0.50$ and $\alpha = 0^\circ$.

Figure. Pressure contours plotted on the surface triangular meshes of a sequence of four successively refined tetrahedral grids.

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Unit time (microsecond)</th>
<th>Speedup vs. CPU-1</th>
<th>Speedup vs. CPU-16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPU-1</td>
<td>CPU-1</td>
<td>CPU-16</td>
</tr>
<tr>
<td>2,426</td>
<td>20.2</td>
<td>176.8</td>
<td>14.8</td>
</tr>
<tr>
<td>16,467</td>
<td>10.7</td>
<td>182.8</td>
<td>12.6</td>
</tr>
<tr>
<td>124,706</td>
<td>9.3</td>
<td>182.8</td>
<td>13.0</td>
</tr>
<tr>
<td>966,497</td>
<td>8.8</td>
<td>198.9</td>
<td>13.1</td>
</tr>
</tbody>
</table>
Example 1. Subsonic Flow past a Sphere

Subsonic flow past a sphere at $M_\infty = 0.50$ and $\alpha = 0^\circ$.

Unit running time vs. Nr. of elements

Speedup vs. Nr. of elements
Example 1. Subsonic Flow past a Sphere

Subsonic flow past a sphere at $M_\infty = 0.50$ and $\alpha = 0^\circ$.

Anim. Comparison of runtime performance rendered by pressure contours on the surface meshes.
Example 2. Transonic Flow over a Boeing 747

Transonic flow over a Boeing 747 aircraft at $M_∞ = 0.85$ and $\alpha = 2°$

Figure. Mach number contours plotted on the surface triangular meshes

<table>
<thead>
<tr>
<th>Nelem</th>
<th>Unit time (microsecond)</th>
<th>Speedup vs. CPU-1</th>
<th>Speedup vs. CPU-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>253,577</td>
<td>11.5 11.5</td>
<td>21.2</td>
<td>1.43</td>
</tr>
<tr>
<td>1,025,170</td>
<td>10.6 10.6</td>
<td>24.5</td>
<td>1.57</td>
</tr>
</tbody>
</table>
Example 2. Transonic Flow over a Boeing 747

Transonic flow over a Boeing 747 aircraft at $M_\infty = 0.85$ and $\alpha = 2^\circ$

Unit running time vs. Nr. of elements

Speedup vs. Nr. of elements
Example 2. Transonic Flow over a Boeing 747

Transonic flow over a Boeing 747 aircraft at $M_\infty = 0.85$ and $\alpha = 2^\circ$

Anim. Comparison of runtime performance rendered by Mach number contours on the surface meshes

- GPU-1
- CPU-16
- CPU-1
**Example 3. Viscous Flow past a Sphere**

Subsonic flow past a sphere at $M_\infty = 0.5$ and $Re = 118$

<table>
<thead>
<tr>
<th>Nelem</th>
<th>GPU-1</th>
<th>CPU-1</th>
<th>CPU-16</th>
<th>vs. CPU-1</th>
<th>vs. CPU-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>200,416</td>
<td>14.6</td>
<td>259.9</td>
<td>20.5</td>
<td>17.8</td>
<td>1.41</td>
</tr>
<tr>
<td>925,995</td>
<td>13.9</td>
<td>257.2</td>
<td>20.6</td>
<td>18.5</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Triangular surface meshes

Streamtraces on the symmetry plane
Example 3. Viscous Flow past a Sphere

Subsonic flow past a sphere at $M_\infty = 0.5$ and $Re = 118$

Anim. Comparison of runtime performance rendered by Mach number contours along with streamtraces on the symmetry plane
Example 4. Large Eddy Simulation (to be completed soon…)

Lid-driven cubical cavity at $M_b = 0.2$ and $Re = 10,000$

- Instantaneous Mach number iso-surfaces in the cubical domain
- Instantaneous Mach number contours on spanwise mid-plane

Simulation of incompressible flows using a compressible flow solver!
## Concluding Remarks

### RDGFLO with OpenACC

- **Performance on single GPU**
  - An average scaling factor of over 20.0x vs. 1 CPU core
  - An average scaling factor of over 1.4x vs. 16 CPU cores

- **Extensibility**
  - The current parallel strategy can be applied to more functionalities

- **Verified compilers**
  - PGI Fortran

- **Verified platform**
  - NVIDIA CUDA-enabled GPUs

- **Limitation**
  - Memory constraint on GPU device
  - Balance between [optimal GPU parallelism] and [code portability]

- **Publications**
Future Work & Prospect

1. Multi-GPU parallelization via MPI
   • Face grouping algorithm for balanced load over partition-domain regions

Physical boundary faces  Partition boundary faces  Internal faces

Now: simple grouping, e.g., in 2 partition domains

<table>
<thead>
<tr>
<th>Physical boundary faces</th>
<th>Partition boundary faces</th>
<th>Internal faces</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 groups</td>
<td>3 groups</td>
<td>6 groups</td>
</tr>
<tr>
<td>4 groups</td>
<td>3 groups</td>
<td>5 groups</td>
</tr>
</tbody>
</table>

Future: balanced-load grouping, e.g., in 2 partition domains

<table>
<thead>
<tr>
<th>Physical boundary faces</th>
<th>Partition boundary faces</th>
<th>Internal faces</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 groups</td>
<td>3 groups</td>
<td>6 groups</td>
</tr>
<tr>
<td>4 groups</td>
<td>3 groups</td>
<td>6 groups</td>
</tr>
</tbody>
</table>
Future Work & Prospect

1. Multi-GPU parallelization via MPI (cont.)

- **Atomic operation: !$acc atomic**
  - Declared in OpenACC 2.0
  - Not yet implemented in PGI Accelerator
  - Why we need it?
    - Significant simplification of programming
    - Potential huge improvement of scalability

---

<table>
<thead>
<tr>
<th>Physical boundary faces</th>
<th>Partition boundary faces</th>
<th>Internal faces</th>
</tr>
</thead>
</table>

Future: atomic operation (no grouping needed), e.g., in 2 partition domains

---

![Boundary Faces Diagram](image-url)
Future Work & Prospect

2. Implicit time integration

- Preconditioned linear solver, e.g., GMRES-LU+SGS algorithm
  - The inverse of the approximate block diagonal Jacobian matrix is the least we need for preconditioning
  - Any direct algorithm of matrix inverse requires some auxiliary arrays, as below

<table>
<thead>
<tr>
<th>OpenMP version</th>
<th>OpenACC version</th>
</tr>
</thead>
<tbody>
<tr>
<td>real*8 A(20,20,Nel)</td>
<td>real*8 A(20,20,Nel)</td>
</tr>
<tr>
<td>real*8 B(20,20)</td>
<td>real*8 B(20,20)</td>
</tr>
<tr>
<td>real*8 C(20)</td>
<td>real*8 C(20)</td>
</tr>
<tr>
<td>!$omp parallel</td>
<td>!$acc parallel</td>
</tr>
<tr>
<td>!$omp do</td>
<td>!$acc loop</td>
</tr>
<tr>
<td>do ie = 1, Nel</td>
<td>do ie = 1, Nel</td>
</tr>
<tr>
<td>!... Direct algorithm to invert A(20,20,ie)</td>
<td>!... Direct algorithm to invert A(20,20,ie)</td>
</tr>
<tr>
<td>!... with the aid of B(20,20) and C(20)</td>
<td>!... with the aid of B(20,20) and C(20)</td>
</tr>
<tr>
<td>enddo</td>
<td>enddo</td>
</tr>
<tr>
<td>!$omp end parallel</td>
<td>!$acc end parallel</td>
</tr>
</tbody>
</table>

However, the direct algorithm in the OpenACC ver. is practically useless, due to very limited “shared memory” (48~64KB) / thread block on GPU!
## Future Work & Prospect

### 2. Implicit time integration (cont.)

- **Solution: iterative algorithm for matrix inverse**
  - **Pros**
    - No need of auxiliary arrays, suitable for OpenACC parallel regions
    - Adequate solution efficiency for diagonal-dominant matrix
  - **Cons**
    - To be explored…

### OpenMP version

```fortran
real*8 A(20,20,Nel)
real*8 B(20,20)
real*8 C(20)
!
$omp parallel
$omp do
  do ie = 1, Nel
    !... Direct algorithm to invert A(20,20,ie)
    !... with the aid of B(20,20) and C(20)
  enddo
$omp end parallel
```

### OpenACC version

```fortran
real*8 A(20,20,Nel)
!
$acc parallel
$acc loop
  do ie = 1, Nel
    !... Iterative algorithm to invert A(20,20,ie)
    !... with no need of auxiliary arrays
  enddo
$acc end parallel
```
Development Plans

Recent work

Multi-GPU parallelization via MPI
- A balanced face grouping algorithm is required over domain partitions.

Challenges for implementing implicit time integration
- Algorithms suitable for GPU parallel computing
- Memory limitation on GPU devices.

Goals

A portable, GPU-accelerated high-order CFD toolkit
- A complete framework for the reconstructed discontinuous Galerkin (RDG) method for compressible flows on hybrid grids

Impact

A potential, competitive parallel-computing model of CFD programming for the next-generation HPC hardware/software