Future Generation Computer Systems 139 (2023) 1–16

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Hybrid MPI and CUDA paralleled finite volume unstructured CFD simulations on a multi-GPU system

Xi Zhang [a,*], Xiaohu Guo [b], Yue Weng [a], Xianwei Zhang [a], Yutong Lu [a], Zhong Zhao [c]

[a] School of Computer Science and Engineering, Sun Yat-sen University, No. 132 East Outer Ring Road, Guangzhou, 510006, Guangdong, China
[b] Hartree Center, STFC Daresbury Laboratory, Keckwick Lane, Warrington, WA4 4AD, England, United Kingdom
[c] Computational Aerodynamics Institute, China Aerodynamics Research and Development Center, No. 6, South Section, Second Ring Road, Mianyang, 621000, Sichuan, China

ARTICLE INFO

Article history:
Received 31 October 2021
Received in revised form 30 August 2022
Accepted 3 September 2022
Available online 12 September 2022

Keywords:
Computational fluid dynamics
Unstructured mesh
Compressible flow
Graphic processing units
Optimizations
Scalability

ABSTRACT

Porting unstructured Computational Fluid Dynamics (CFD) analysis of compressible flow to Graphics Processing Units (GPUs) confronts two difficulties. Firstly, non-coalescing access to the GPU's global memory is induced by indirect data access leading to performance loss. Secondly, data exchange among multi-GPU is complex due to data communication between processes and transfer between host and device, which degrades scalability. For increasing data locality on unstructured finite volume GPU simulations for compressible flow, we perform some optimizations, including cell and face renumbering, data dependence resolving, nested loops split, and loop mode adjustment. Then, a hybrid MPI-CUDA parallel framework with packing and unpacking exchange data on GPU is established for multi-GPU computing. Finally, after optimizations, the performance of the whole application on a GPU is increased by around 50%. Simulations of ONERA M6 cases on a single GPU (Nvidia Tesla V100) can achieve an average of 13.4 speedup compared to those on 28 CPU cores (Intel Xeon Gold 6132). On the baseline of 2 GPUs, strong scaling results show a parallel efficiency of 42% on 200 GPUs, while weak scaling tests give a parallel efficiency of 82.4% up to 200 GPUs.

1. Introduction

Unstructured mesh CFD approaches [1] are widely applied in aerospace science and engineering for simulating compressible flow-solved physical scenarios. Moreover, mesh generation is more convenient for the computational domain with complex geometry [2]. Hence, many unstructured CFD software is developed, such as Fun3D [3], SU2 [4], OpenFOAM [5], Fluidity [6], NNW-PHengLEI [7], Quinoa [8], etc. In recent years, multi-core computing hardware has developed fast. The GPU has played a more crucial role in CFD, owning to its high parallel computing capacity and low energy consumption [9]. However, it is difficult for unstructured GPU simulations to achieve high performance.

To illustrate, data storage on the unstructured mesh is irregular, which leads to indirect data access [10]. Indirect data access induces the non-coalescing loads and stores on the GPU. Due to the special memory access mode of the GPU [11], non-coalescing data access results in significant performance loss. On the other hand, data exchange between partitioned domains is necessary for multi-GPU simulations, which require both data communication and host-device data transfer [12]. The complicated data exchange mechanism may degrade parallel efficiency.

Renumbering cells and faces in the mesh can be applied to reduce non-coalescing memory access latency on the GPU. Most studies concentrate on renumbering schemes for implicit methods such as the LU-SGS method [13]. In those works, renumbering algorithms such as the Reverse Cuthill-Mckee (RCM) [14] reduced the bandwidth of the matrix. Corrigan et al. [15] renumbered cells and faces in high-order explicit Discontinuous Galerkin Euler solver on an unstructured mesh. A line-based renumbering scheme [16] is proposed based on topological information, including cells, faces, and nodes. Lani et al. applied the RCM algorithm in unstructured finite volume simulations [17]. However, performance improvement is not obvious. Therefore, an efficient renumbering scheme is meaningful for explicit finite volume simulations on the GPU.

The data race problem induced by many threads writing simultaneously in the same global memory often exists in GPU computing on the unstructured mesh. Atomic operations and graph coloring are methods to address the data race problem based on hardware and software, respectively. In the CPU-only environment, some researchers [18]found that both general and optimized graph coloring methods are better than atomic operations. On the GPU, some studies indicated that optimized graph coloring beats atomic operations in unstructured FEM [19], FVM [20] applications, and some benchmarks [21]. However, only

* Corresponding author.
E-mail address: zhangx299@mail.sysu.edu.cn (X. Zhang).

https://doi.org/10.1016/j.future.2022.09.005
0167-739X/© 2022 Elsevier B.V. All rights reserved.

simple 2D and 3D meshes are considered in those works. With the development of hardware, atomic operations are optimized on the GPU [22]. Hence, comparison of graph coloring and atomic operations is crucial in unstructured 3D typical industry meshes on the GPU.

A suitable data layout can improve the data locality. Structure of Array (SOA) data layout often results in contiguous data access [23]. However, some work showed that Array of Structure (AOS) makes good use of cache [24]. Besides data layouts, data locality can also be improved by loop adjustment. Some researchers indicated that running separate concurrent kernels can make good use of GPU resources [25]. Moreover, different loop modes may be used in the same kernel. A suitable mesh loop may increase the data locality. Therefore, it is significant to optimize data locality by adjusting mesh loops.

Most multi-GPU CFD simulations are designed only to target GPUs, where CPUs are used for managing GPUs. Load balance is achieved easily in GPU-only mode, which makes multi-GPU computing more efficient. Jacobsen et al. [26] developed a hybrid MPI-CUDA paralleled 3D incompressible solver. The research showed that it is hard to overlap GPU computing, host-device data transfer, and MPI communication using more than 16 GPUs. Zolfaghari et al. [27] used a hybrid MPI-CUDA parallel framework for high order 3D incompressible Direct Numerical Simulations. Without hide of device copies and MPI communication, the scalability goes down using more than 32 GPUs. Vincent et al. developed a hybrid MPI-CUDA paralleled framework in a high-order unstructured CFD solver called pyFR based on the flux reconstruction method. GPU-only mode was used in most simulations [28]. Similarly, some researchers [29,30] chose GPU-only mode in the flux reconstruction method. CUDA-aware MPI support is applied to reduce overheads by data exchange. Oyarzen et al. [31] developed a portable MPI-CUDA paralleled CFD model for CPU/GPU heterogeneous supercomputer. It is found that GPU-only mode owns better performance but worse scalability than CPU-only mode. A few researchers applied CPU–GPU mode in heterogeneous computing, where both CPUs and GPUs are used to compute. CPU computing should not be neglected, if the speedup of a GPU compared to a CPU is small. Álvarez-Farré et al. [32] established an MPI-OpenMP-OpenCL paralleled framework. It is indicated that communication overlapping with computation is hard on Nvidia Tesla V100. Borrel et al. [33]used a hybrid MPI-OpenMP-CUDA parallel framework for simulating incompressible flow over an airplane on a CPU–GPU heterogeneous architecture with POWER9 and Nvidia Tesla V100. CPUs are used to compute given the small speedup of a GPU to a CPU. Thus, it is crucial to study the effects of the parallel framework on the performance of multi-GPU simulations.

In this paper, an unstructured finite volume cell centered explicit CFD solver is ported on a multi-GPU computing system. Two contributions are made. Firstly, a series of strategies are applied for optimizing indirect data access. Specifically, a face renumbering strategy is proposed for keeping data close globally in a finite volume solver, which is implemented easily with only cell-face topological information. Then, atomic operations and the general graph coloring method are compared in GPU kernels with race conditions. Typical industry 3D meshes are used, which is different from some work on simple meshes [19–21]. Then, loop adjustment strategies are applied to enhance data locality without the change of data layouts in some studies [23,24]. Those optimizations made performance improve significantly on both kernel and application levels. Secondly, a hybrid MPI-CUDA parallel framework is established and evaluated on a multi-GPU computing system. Packing and unpacking communication data on the GPU make both performance and scalability grow up.

The remaining paper contains the following parts: Section 2 describes the mathematical model. Section 3 introduces optimization strategies. Section 4 shows a hybrid MPI-CUDA paralleled framework and packing/unpacking exchange data on the GPU. Section 5 presents the test environment. Section 6 validates GPU simulation results. Section 7 shows the effects of optimizations on both kernel and application levels. Section 8 evaluates the parallel framework. Section 9 concludes the performance of the hybrid MPI-CUDA parallel framework.

2. Mathematical model

The research is performed on a CFD suite of tools called NNW-PHengLEI developed by China Aerodynamic R&D Center [7]. NNW-PHengLEI's purpose is to study and develop algorithms for modeling compressible flow. NNW-PHengLEI contains both structured and unstructured solvers. Only a transonic flow solver on the unstructured mesh is used in the research.

2.1. Governing equations

The motion of fluid can be described by Navier–Stokes (NS) equations. The integral form of NS equations is shown in Eq. (1).

$$\frac{\partial}{\partial t}\int_V Q dV + \oint_{\partial V}(F_c - F_v)dS = 0 \tag{1}$$

In the control volume V with boundary dV, five unknowns in vector (Q) form are described in Eq. (2), including the density ρ, velocity components u, v, w, and the internal energy e.

$$Q = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix} \tag{2}$$

F_c represents the convective flux, shown in Eq. (3). U is described by $U = n_x u + n_y v + n_z w$. n_x, n_y, and n_z are components of the unit normal vector on dV.

$$F_c = \begin{bmatrix} \rho U \\ \rho u U + n_x p \\ \rho v U + n_y p \\ \rho w U + n_z p \\ \rho U(e + \frac{u^2+v^2+w^2}{2} + \frac{p}{\rho}) \end{bmatrix} \tag{3}$$

$$F_v = \begin{bmatrix} 0 \\ n_x\tau_{xx} + n_y\tau_{xy} + n_z\tau_{xz} \\ n_x\tau_{yx} + n_y\tau_{yy} + n_z\tau_{yz} \\ n_x\tau_{zx} + n_y\tau_{zy} + n_z\tau_{zz} \\ n_x\theta_x + n_y\theta_y + n_z\theta_z \end{bmatrix} \tag{4}$$

F_v means the viscous flux, described in Eq. (4). F_v is computed by the unit normal vector (n_x, n_y, and n_z), the viscous stress tensor (τ_{xx}, …, and τ_{zz}), and energy dissipation terms including θ_x, θ_y, and θ_x. Both viscous stress tensor and energy dissipation terms require the gradient of Q and some physical coefficients such as the temperature T, the laminar viscous coefficient, the turbulent viscous coefficient, etc.

The finite volume method is applied in the numerical simulation. The fluid domain is discretized into a series of cells (also called control volumes). Five independent unknown variables are cell-based. The boundary of cells are called faces. On a cell with volume V_{cell}, owning N_f faces with area $S_m (m = 1, \dots, N_f)$, NS equations can be discretized, as described by,

$$\frac{\Delta Q_{cell}}{\Delta t} = -\frac{1}{V_{cell}}\left[\sum_{m=1}^{N_f}(F_c - F_v)_m S_m\right] \tag{5}$$

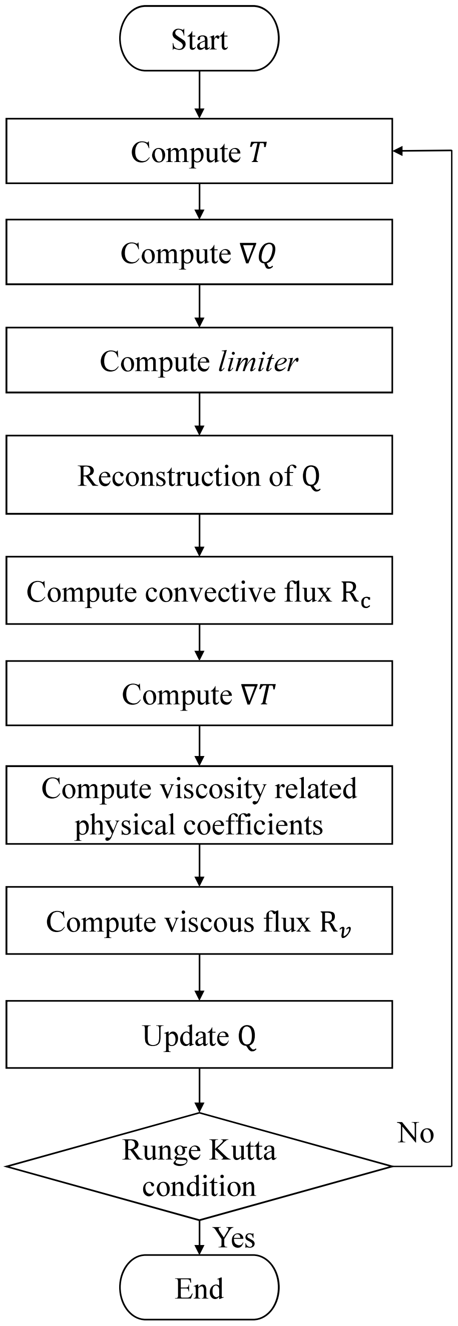


Fig. 1. Computing procedures for NS equations in one iteration step.

In Eq. (5), Roe scheme is applied to avoid flow oscillation in computing the convective flux F_c. The viscous flux F_v is calculated by the central scheme. In explicit numerical simulations, iteration is necessary to guarantee the residual of ΔQ_{cell} is small enough. In an iteration step, the temporal term $\frac{\Delta Q_{cell}}{\Delta t}$ is computed by the two-step Runge–Kutta method. The Spalart–Allmaras turbulence model is applied for the turbulence effects. The treatment of the turbulence equation is similar to NS equations.

2.2. Computing procedure and formulas

The computing procedure of NS equations is described in Fig. 1. In each iteration step, Δt is determined by the CFL number. One time step is divided into two small ones by the two-step Runge–Kutta scheme. Then, the convective flux is computed with

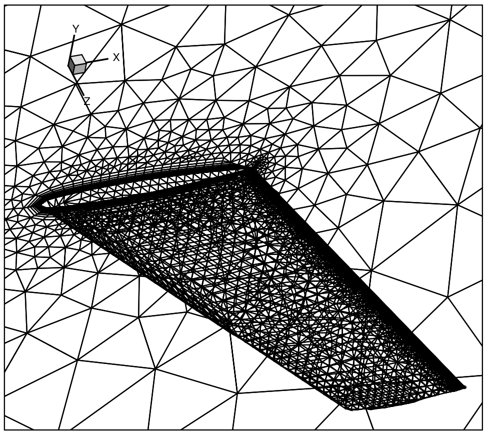


Fig. 2. Unstructured mesh in cell-centered finite volume method.

T, the gradient of Q (∇Q) and T (∇T), limiter, and reconstruction of Q on faces. Thirdly, the viscous flux is computed by Q, ∇Q, and some other viscosity-related coefficients on faces. Finally, Q_{cell} is updated with F_c and F_v. Some formulas in the computing procedure are crucial for porting and optimizing the computing procedure on the GPU.

Interpolation of Q from cell centers into nodes is required to compute ∇Q_{cell}, as described by

$$Q_{node} = \frac{1}{N_{cell}} \left[\sum_{n=1}^{N_{cell}^{node}} (Q_{cell})_n \right] \tag{6}$$

where Q_{node} based on nodes is the average of Q_{cell} based on cells, and N_{cell}^{node} is the number of cells sharing the same node. Similarly, computing ∇T_{cell} requires interpolating T from cells to nodes.

For computing limiter, the local maximum pressure p_{cell}^{max} and the local minimum pressure p_{cell}^{min} should be computed by

$$p_{cell}^{max} = max(p_{cell}^{n}, n = 0 \ldots N_{cell}^{cell}) \tag{7}$$

$$p_{cell}^{min} = min(p_{cell}^{n}, n = 0 \ldots N_{cell}^{cell}) \tag{8}$$

where p_{cell}^{0} is the pressure on the local cell, and $p_{cell}^{n}(n = 1, \ldots, N_{cell}^{cell})$ is the pressure on the neighbor cells. N_{cell}^{cell} is the amount of the neighbor cells.

The summation of *flux* (F_c or F_v) located on faces is required to update Q, as described by

$$res_{cell} = \sum_{m=1}^{N_f} flux_f^m \cdot S_m \tag{9}$$

where res_{cell} stores the summation of *flux*, which are cell-centered.

For the turbulence equation, computing procedures are similar.

2.3. Unstructured mesh connectivity

In the Finite Volume Method, the physical domain is discretized into many non-overlapping arbitrary polyhedral control volumes. The discretization of the fluid domain of the ONERA M6 wing is described in Fig. 2. An unstructured mesh is formed by those control volumes, also called cells. The boundary of cells is called faces. Faces' vertices are called nodes. Fig. 3 shows some hexahedral cells with faces and nodes.

Each cell, face, and node owns a unique index in an unstructured mesh. Then, data located on cells, faces, and nodes are stored in the order of those indices. Due to unstructured

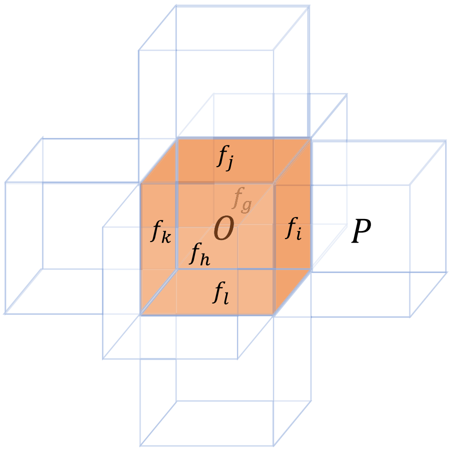


Fig. 3. The unstructured mesh in cell-centered finite volume method.

property, cells, faces, or nodes with adjacent indexes do not mean connectivity in the physical domain. Thus, unstructured mesh connectivity between cells, faces, and nodes should be defined with indices especially. In NNW-PHengLEI, the physical connectivity is defined by cell-face, cell-node, and face-node connectivity.

To explain the unstructured mesh connectivity and data storage clearly, variables in the form of scalar will be standardized, while those in the form of the array will be italic.

Cell-Face connectivity is described by *leftCell*, *rightCell*, and *cellFaces*. Cell indices are stored in both *leftCell* and *rightCell* in the order of face indices. To illustrate, in Fig. 3, face f_i is owned by both cell O and cell P. Hence, the cell-face connectivity between cell O, cell P, and face f_i can be described by $leftCell[f_i] = O$ and $rightCell[f_i] = P$. Besides face f_i, cell O also has the other faces, including f_j, f_k, f_l, f_h, and f_g. The cell-face connectivity between cell O, faces f_g, f_h, f_i, f_j, f_k, and f_l is described by $cellFace[offsetCellFace[O]+n] = f_g, f_h, f_i, f_j, f_k, f_l (n = 1, \dots, numFaceOfCell[O])$. Because all unstructured mesh connectivity information is in one-dimension array, an offset of O in *cellFace* ($offsetCellFace[O]$) is required to traverse faces belonging to O. Apart from $offsetCellFace[O]$, $numFaceOfCell[O]$ stores the number of faces belonging to O, which is 6 in the case.

Cell-Cell connectivity can be obtained by cell-face connectivity. According to *leftCell*, *rightCell*, and *cellFace*, *cellCells* obtained and stores cells' neighbor cell indices. The 1D array should be used with *offsetCellCells* and *numCellOfCell*.

Face-Node connectivity is defined by two 1D arrays including *faceNodes* and *nodeFaces*. *faceNodes* focuses on faces and stores nodes on faces. Similar to *cellFaces*, *faceNode* is used with *offsetFaceNode* and *numNodeOfFace* for accessing all nodes on a face. Similarly, *nodeFaces* owns faces sharing the same node with the help of *numFacesInNode* and *faceOfNodeStart*.

Cell-Node connectivity can be defined by *cellNodes* and *nodeCells*, similar to face-node connectivity.

2.4. Data structure and storage

The SOA data layout is applied in both host and device codes. Therefore, those data can be accessed directly by indices of cells, faces, and nodes. According to GPU data Load/Store mode, the direct accessing of data is the best way to make use of the GPU. However, indirect data access exists widely in computing procedures when more than one type of data on cells, faces, or nodes exists in a computing loop.

For example, in the flux summation (Eq. (9)), the cell data *res* is updated by the sum of the face data *flux*. In Algo. 1, a face loop (Line 1) is used to update *res* on every cell. So, *flux* is loaded directly by face indices (faceID). On the contrary, *res* is stored indirectly by cell indices (Le and Re) with *leftCellOfFace* and *rightCellOfFace*. Finally, *res* is added with *flux* (Line 5–6) by the operation **setAdd**.

Algorithm 1 Summation of Flux by a face loop (SF-CPU)

```
1: for faceID = nBoundFace to nTotalFace-1 do
2:   Le ← leftCellOfFace[faceID]
3:   Re ← rightCellOfFace[faceID]
4:   for eqnID = 0 to numEqn - 1 do
5:     setAdd(res[eqnID][Le], flux[eqnID][faceID])
6:     setAdd(res[eqnID][Re], flux[eqnID][faceID])
7:   end for
8: end for
```

Indirect data access will induce data locality and data dependence problems on the GPU. Firstly, indirect data access often cannot guarantee data coalescing on the GPU. For example, in the face loop induced indirect memory access of *res*, both $res[leftCell[i]]$ and $res[rightCell[i]]$ are not continuous in global memory, as the index i increases from 0 to the total face number nTotalFace. The non-coalescing data access induces remarkable overheads. Secondly, data dependence is often induced by indirect data access as well. Specifically, many GPU threads may write in the same shared or global memory instantly. For example, in the flux summation, some threads on the face loop may instantaneously access the same space of *res* by $res[leftCell[i]]$ or $res[rightCell[i]]$, which leads to incorrect results. Therefore, optimizations and special treatment should be performed for efficient and precise simulations.

3. Algorithms and optimizations

After the NNW-PHengLEI program is ported on the GPU, the initial performance is evaluated and analyzed. Then, several optimization strategies are used to reduce executing time on the GPU. Specifically, cell and face renumbering, data dependence resolving, nested loops split, and loop mode optimization are applied for optimizing memory access globally or locally.

3.1. Hot spots analysis

CUDA C is applied for porting NNW-PHengLEI program to the GPU. All computing procedures in Fig. 1 are ported to the GPU to avoid frequent host-device data transfers. Host-to-Device (HtoD) data input only exists before iterations, and Device-to-Host (DtoH) only happens once as all iteration steps finish.

The performance of the NNW-PHengLEI CUDA program is evaluated by NVPROF, on a moderate mesh (3.97 million cells and 8.83 million faces), for the simulation of external flow over the ONERA M6 wing. Table 1 shows the time fraction of 10 hot spots indicating that those hot spots account for almost 80% of total execution time.

From the description of those hot spots, it can be seen that indirect memory access exists in all hot spots due to data types and loop modes. Non-coalescing data access may be eased by rearranging data globally. The race condition is induced by indirect data access, which accounts for more than half of the total executing time. Thus, an efficient method to resolve the race condition is significant for GPU computing. Furthermore, nested loops exist in most hot spots, which take more than 40% of the total executing time. So, it is crucial to adjust nested loops to reduce overheads induced by indirect memory access. Thirdly, only the face loop is used in all hot spots. The other loop modes may optimize data locality.

Table 1
Description of kernels.

No.	Description	Race condition	Nested loop	Loop mode	FLOP	Cell data fp64	Cell data int32	Face data fp64	Face data int32	Node data fp64	Node data int32	Time fraction
K1	∇Q and ∇T computing	yes	no	face	16	12	0	4	6	3	1	21.34%
K2	Interpolation of Q and T	yes	yes	face	36	36	0	0	6	18	3	11.98%
K3	Viscous flux computing	no	yes	face	310	48	0	26	2	0	0	11.40%
K4	*limiter* computing	yes	no	face	40	20	0	3	2	0	0	7.41%
K5	Face value reconstruction	no	yes	face	86	46	0	13	2	0	0	7.39%
K6	Flux summation (NS)	yes	yes	face	10	20	0	5	2	0	0	6.89%
K7	Get face value from Q	no	yes	face	0	10	0	10	2	0	0	3.35%
K8	Local pressure comparing	yes	no	face	4	10	0	0	3	0	0	2.89%
K9	Turbulence coefficients	yes	yes	face	97	26	0	0	0	6	2	2.65%
K10	Flux summation (Turbulence)	yes	no	face	16	16	0	0	0	2	2	2.42%

3.2. Cell and face renumbering

Data in the unstructured mesh is stored in the order of cells, faces, or nodes. Hence, renumbering cells and faces may improve data storage globally.

Firstly, to reduce the bandwidth (index jump) between cells and their neighbor cells, the Reverse Cuthill-Mckee (RCM) [14] algorithm is applied for reordering cell indices and the corresponding mesh connectivity information related to cells such as *cellFace*, *cellNode*, and *cellCell*, etc. Then, after cell renumbering, faces are also renumbered. In the NNW-PHengLEI program, boundary faces are gathered together and numbered from 0 to nBoundFace-1. Interior faces are renumbered according to the updated Cell-Face connectivity information *cellFace* described in Algo. 2.

Interior faces are reordered so that the face index (faceID) is in ascending order in *cellFace* shown in Line 1. Then, a cell loop is applied for traversing *cellFace* in Line 4. In the condition of faceID>nBoundFace-1 (Line 9), only interior faces are renumbered. The face index in labelFace starts from nBoundFace-1 in Line 3 and increases if only a face index in faceID is not set (Line 10–11). Finally, old and new face indices are mapped in the array *mapFace* updated by the face index (labelFace) in Line 12. With the help of *mapFace*, the mesh connectivity information related to faces should be updated by new face indices, such as *cellFace*, *faceCell*, and *faceNode*, etc.

Algorithm 2 Face renumber

```
1:  Reorder faces in cellFace by ascending order
2:  mapFace ← -1
3:  labelFace ← nBoundFace-1
4:  for cellID = 0 to nTotalCell-1 do
5:      offset ← offsetCellFace[cellID]
6:      numFaces ← numFaceOfCell[cellID]
7:      for faceInCell = 0 to numFaces-1 do
8:          faceID ← cellFace[offset+faceInCell]
9:          if faceID > nBoundFace-1 then
10:             if mapFace[faceID] == -1 then
11:                 labelFace ← labelFace+1
12:                 mapFace[faceID] ← labelFace
13:             end if
14:         end if
15:     end for
16: end for
17: Update mesh connectivity information by mapFace
```

3.3. Race condition resolving

As kernels are executed on a GPU, many threads may write data in the same global or shared memory, which induces wrong results. An example is shown in Algo. 1. The race condition can be tackled by both software and hardware-based methods. It is significant to investigate which one is suitable for GPU simulations in unstructured meshes.

The graph coloring method is software-based, which divides GPU computing on different threads into several groups. In each group, no global or shared memory is updated by more than one thread. A graph coloring method for resolving the race condition in the flux summation is shown in Algo. 3. Faces are divided into some groups to avoid the race condition. Hence, kernels are launched several times (nGroups) in Line 2. In each face group, the size (numFacesInGroup) and the start position (groupStart) are obtained firstly in Line 3–4. Then, a face-based loop (numFacesInGroup) is applied for a face group in Line 7. Those faces distributed to threads are determined by the array *faceGroup* in Line 9. Afterward, because faces belonging to one cell is not in the same face group, cell data *res* can be updated simultaneously by many threads ported on faces. Thus, the operation **setAdd** used on the CPU (Algo. 1) can be applied directly to the GPU, as well. Furthermore, *flux* is reordered by face indices in *faceGroup* (Line 1), leading to a continuous load of *flux* (Line 13–14).

Algorithm 3 Summation of Flux by graph coloring(SF-GC)

```
1:  Reorder flux according to faceGroup
2:  for groupID = 0 to nGroups-1 do
3:      numFacesInGroup←numFaceOfGroup[groupID]
4:      groupStart←offsetFaceGroup[groupID]
5:      <GPU kernel Begin>
6:      threadID←threadIdx.x+blockIdx.x*blockDim.x
7:      for faceGroupID = threadID to numFacesInGroup-1 do
8:          groupFaceID←groupStart+faceGroupID
9:          faceID←faceGroup[groupFaceID]
10:         Le←leftCellOfFace[faceID]
11:         Re←rightCellOfFace[faceID]
12:         for eqnID = 0 to numEqn - 1 do
13:             setAdd(res[eqnID*nTotalCell+Le],
                flux[eqnID*nTotalFace+faceID])
14:             setAdd(res[eqnID*nTotalCell+Re],
                flux[eqnID*nTotalFace+faceID])
15:         end for
16:         setAdd(faceGroupID, blockDim.x*gridDim.x)
17:     end for
18:     <GPU kernel End>
19: end for
```

The atomic operation is hardware-based, which is achieved by the GPU-supported thread-lock mechanism and CUDA-supported API. By thread-lock, no thread can write in the same global or shared memory. Algo. 4 describes atomic operations used in the summation flux. All interior faces are ported on computing threads by a face loop. The CUDA-supported API **atomicAdd** is applied to assign the summation of *flux* and *res* into *res* in Line 7–8. It guarantees that the same element in *res* can only be stored by a computing thread once.

3.4. Nested loops split

Nested loops exist widely in NNW-PHengLEI. Most GPU computing is related to the loop on geometry, including cells, faces, or nodes. The other loop related to data dimension is often required. Many data related to different physical meanings or dimensions of space own several dimensionalities. For example, in Algo. 1, arrays *res* and *flux* own five dimensions (numEqn), corresponding to different unknowns in Q shown in Eq. (2). Those dimensions should be operated by a loop in Line 4. of Algo. 4. Line 6–9 shows that although variables *flux* and *res* in GPU global memory are stored in 1D arrays, a dimension loop is still necessary for traversing all elements.

Algorithm 4 Summation of Flux by atomic operation (SF-AT)

```
1:  <GPU kernel Begin>
2:  threadID ← threadIdx.x+blockIdx.x*blockDim.x
3:  for faceID = nBoundFace+threadID to nTotalFace-1 do
4:      Le ← leftCellOfFace[faceID]
5:      Re ← rightCellOfFace[faceID]
6:      for eqnID = 0 to numEqn - 1 do
7:          atomicAdd(res[eqnID*nTotalCell+Le],
            flux[eqnID*nTotalFace+faceID])
8:          atomicAdd(res[eqnID*nTotalCell+Re],
            flux[eqnID*nTotalFace+faceID])
9:      end for
10:     setAdd(faceID, blockDim.x*gridDim.x)
11: end for
12: <GPU kernel End>
```

The dimension loop can be arranged outside kernels, which may increase the performance. In Algo. 5, GPU kernels are launched several times (numEqn) by a dimension loop in Line 1. Only the face loop in Line 4 is applied in kernels. Finally, a large kernel with nested loops is split into several small ones corresponding to data dimensions.

Algorithm 5 Summation of Flux by split loops (SF-SL)

```
1:  for eqnID = 0 to numEqn - 1 do
2:      <GPU kernel Begin>
3:      threadID ← threadIdx.x+blockIdx.x*blockDim.x
4:      for faceID=nBoundFace+threadID to nTotalFace-1 do
5:          Le ← leftCellOfFace[faceID]
6:          Re ← rightCellOfFace[faceID]
7:          atomicAdd(res[eqnID*nTotalCell+Le],
            flux[eqnID*nTotalFace+faceID])
8:          atomicAdd(res[eqnID*nTotalCell+Re],
            flux[eqnID*nTotalFace+faceID])
9:          setAdd(faceID, blockDim.x*gridDim.x)
10:     end for
11:     <GPU kernel End>
12: end for
```

3.5. Loop mode adjustment

Most computing procedures in CFD are related to the geometry-based loop on cells, faces, nodes, etc. Data is also geometry-based, sorted by indices of cells, faces, nodes, etc. Hence, data may be indirectly accessed by some loop modes such as a cell loop on face data, a face loop on cell data, a face loop on node data, etc. For example, in the flux summation (Algo. 1), cell data *res* is indirectly stored (Line 5–6) by a face loop (Line 1). Overheads induced by indirect data access are aggregated on the GPU due to the memory access mechanism.

3.5.1. Data interpolation

In the NNW-PHengLEI program, most hot spots are based on the face loop. However, face data does not exist in some kernels, including data interpolation (Algo. 7) and local pressure comparing (Algo. 10). Indirect data access is induced in those kernels. In some conditions, more than one loop mode can be used for computing. An appropriate loop mode can avoid indirect data access. Therefore, it is crucial to find suitable loop modes, to reduce overheads by indirect data access.

Data interpolation described in Eq. (6) from cells to nodes is required to compute the gradient of Q and T (∇Q, and ∇T). Algo. 6 describes that a face loop (Line 1) is used for data interpolation. Firstly, both left and right cells determined in Line 2–3 own the same face index (faceID). Then, nodes in a face are obtained by face-node information (*faceNode*, *offsetFaceNode*, and *numNodeOfFace*) in Line 4–7. Finally, contributions of cell data q and t to corresponding node data are accumulated in node data *qNode* and *tNode* respectively in Line 8–17. Similarly, the amount of cells sharing the same node is accumulated in the node data *nCount*. In data interpolation, a face loop (Line 1), a node loop (Line 6), and two dimension loops (Line 8, 13) are nested for traversing nodes on faces with dimensions of space.

Algorithm 6 Data interpolation on CPU (DI-CPU)

```
1:  for faceID = nBoundFace to nTotalFace-1 do
2:      Le ← leftCellOfFace[faceID]
3:      Re ← rightCellOfFace[faceID]
4:      faceNodeStart ← offsetFaceNode[faceID]
5:      numNodeInFace ← numNodeOfFace[faceID]
6:      for faceNodeID = 0 to numNodeInFace-1 do
7:          nodeID ← faceNode[faceNodeStart+faceNodeID]
8:          for eqnID = 0 to numEqn - 1 do
9:              setAdd(qNode[eqnID][nodeID], q[eqnID][Le])
10:         end for
11:         setAdd(tNode[nodeID], t[Le])
12:         setAdd(nCount[nodeID], 1)
13:         for eqnID = 0 to numEqn - 1 do
14:             setAdd(qNode[eqnID][nodeID], q[eqnID][Re])
15:         end for
16:         setAdd(tNode[nodeID], t[Re])
17:         setAdd(nCount[nodeID], 1)
18:     end for
19: end for
```

A GPU kernel for data interpolation is described in Algo. 7. It indicates that all faces are ported to computing threads in Line 3. The face-based loop on node data makes many threads may write in the same global memory in Line 11, 13–14, 16, 18–19. Hence, **atomicAdd** resolving the race condition should be used for accumulating contributions of cell data (q and t) to node data (*qNode*, *tNode*, and *nCount*).

For optimizations of data interpolation, cell loops are ported to the GPU as described in Algo. 8 Line 4, 17, and 29. Furthermore, considering nested loops split, a dimension loop in Line 1 is outside of GPU computing. However, indirect data access still exists in those kernels. Can the cell loop result in smaller overheads compared with the face loop?

Algorithm 7 Data interpolation by face loop (DI-F)

```
<GPU kernel Begin>
threadID ← threadIdx.x+blockIdx.x*blockDim.x
for faceID = nBoundFace+threadID to nTotalFace-1 do
    Le ← leftCellOfFace[faceID]
    Re ← rightCellOfFace[faceID]
    faceNodeStart ← offsetFaceNode[faceID]
    numNodeInFace ← numNodeOfFace[faceID]
    for faceNodeID = 0 to numNodeInFace-1 do
        nodeID ← faceNodes[faceNodeStart+faceNodeID]
        for eqnID = 0 to numEqn - 1 do
            atomicAdd(qNode[eqnID*nTotalNode+nodeID],
            q[eqnID*nTotalCell+Le])
        end for
        atomicAdd(tNode[nodeID], t[Le])
        atomicAdd(nCount[nodeID], 1)
        for eqnID = 0 to numEqn - 1 do
            atomicAdd(qNode[eqnID*nTotalNode+nodeID],
            q[eqnID*nTotalCell+Re])
        end for
        atomicAdd(tNode[nodeID], t[Re])
        atomicAdd(nCount[nodeID], 1)
    end for
    setAdd(faceID, blockDim.x*gridDim.x)
end for
<GPU kernel End>
```

3.5.2. Local pressure comparing

Local pressure comparing is applied for determining both local maximum and minimum pressure, shown in Eqs. (7) and (8). In Algo. 9, a face loop in Line 1 can be applied for comparing cell data *pressure* with data (*pMin* and *pMax*) on neighbor cells in Line 4–7. Nevertheless, indirect data access is induced as cell data is traversed by the face loop. Besides that, the race condition is also induced by many threads updating the same global memory for *pMax* and *pMin*. In the kernel shown in Algo. 10, atomic operations in Line 6–9 for comparing, including **atomicMax** and **atomicMin** are used to resolve the data dependence problem.

Because only cell data is used in local pressure comparing, both indirect data access and the race condition can be avoided by the cell loop. In Algo. 11, a cell loop in Line 3 is ported to the GPU. In one cell, the neighbor cells' size (numCell) and the initial position (cellStart) are gained by cell–cell information (*numCellCell* and *offsetCellCell*) in Line 4, 5. Then, neighbor cells are accessed by a loop (Line 6) on *cellCell*. Cell data (*pMin* and *pMax*) are compared with the pressure on neighbor cells by **setCompMin** and **setCompMax** that are the same on the host code (Line 8 and 9).

4. Hybrid MPI and CUDA parallel framework

A hybrid MPI-CUDA parallel framework is established for multi-GPU computing, which data packing and unpacking on the GPU.

4.1. Domain decomposition

For multi-GPU computing, we apply the domain decomposition method. The unstructured mesh is partitioned into many zones by a key-way method in METIS [34]. Each zone is distributed on one GPU that controlled by only one process. Because most computing procedures are on the GPU, the CPU is only used for controlling the GPUs. Except for those CPU cores managing GPUs, the other CPU cores are idle in computing nodes. Fig. 4 shows the relationship between mesh partition, GPUs, and CPUs.

Algorithm 8 data interpolation by cell loop (DI-C)

```
for eqnID = 0 to numEqn - 1 do
    <GPU kernel Begin>
    threadID ← threadIdx.x+blockIdx.x*blockDim.x
    for cellID = threadID to nTotalCell-1 do
        cellNodePosi ← offsetCellNode[cellID]
        for offset = 0 to numNodeOfCell[faceID] - 1 do
            nodeID ← cellNodes[cellNodePosi+offset]
            accessFrequency ← cellNodeCount[cellNodePosi+offset]
            atomicAdd(qNode[eqnID*nTotalNode+nodeID],
            accessFrequency*q[eqnID*nTotalCell+cellID])
        end for
        setAdd(cellID, blockDim.x*gridDim.x)
    end for
    <GPU kernel End>
end for
<GPU kernel Begin>
threadID ← threadIdx.x+blockIdx.x*blockDim.x
for cellID = threadID to nTotalCell-1 do
    cellNodePosi ← offsetCellNode[cellID]
    for offset = 0 to numNodeOfCell[faceID] - 1 do
        nodeID ← cellNodes[cellNodePosi+offset]
        accessFrequency ←
        cellNodeCount[cellNodePosi+offset]
        atomicAdd(tNode[nodeID], accessFrequency*t[cellID])
    end for
    setAdd(cellID, blockDim.x*gridDim.x)
end for
<GPU kernel End>
<GPU kernel Begin>
threadID ← threadIdx.x+blockIdx.x*blockDim.x
for cellID = threadID to nTotalCell-1 do
    cellNodePosi ← offsetCellNode[cellID]
    for offset = 0 to numNodeOfCell[faceID] - 1 do
        nodeID ← cellNodes[cellNodePosi+offset]
        accessFrequency ←
        cellNodeCount[cellNodePosi+offset]
        atomicAdd(nCount[nodeID], accessFrequency)
    end for
    setAdd(cellID, blockDim.x*gridDim.x)
end for
<GPU kernel End>
```

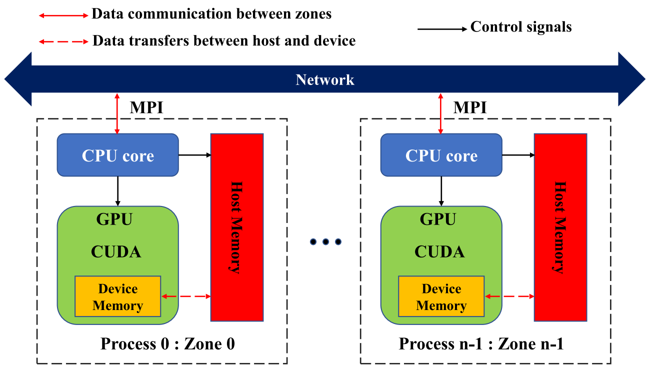


Fig. 4. The relationship between mesh partition (zones), CPUs, and GPUs.

Due to mesh partitioning, the boundary exists between two adjacent zones, which is called an interface. In multi-GPU computing, data exchange on zones' interfaces. Fig. 4 shows two levels in data exchange. In the first level, data is transferred in the same process between GPU and CPU. Secondly, data is communicated via MPI between different processes.

Algorithm 9 Local pressure comparing on CPU (LPC-CPU)

```
1: for faceID = nBoundFace to nTotalFace-1 do
2:   Le ← leftCellOfFace[faceID]
3:   Re ← rightCellOfFace[faceID]
4:   setCompMin(pMin[Le], pressure[Re])
5:   setCompMax(pMax[Le], pressure[Re])
6:   setCompMin(pMin[Re], pressure[Le])
7:   setCompMax(pMax[Re], pressure[Le])
8: end for
```

Algorithm 10 Local pressure comparing by face loop (LPC-F)

```
1: <GPU kernel Begin>
2: threadID←threadIdx.x+blockIdx.x*blockDim.x
3: for faceID = threadID+nBoundFace to nTotalFace-1 do
4:   Le ← leftCellOfFace[faceID]
5:   Re ← rightCellOfFace[faceID]
6:   atomicMin(pMin[Le], pressure[Re])
7:   atomicMax(pMax[Le], pressure[Re])
8:   atomicMin(pMin[Re], pressure[Le])
9:   atomicMax(pMax[Re], pressure[Le])
10:  setAdd(faceID, blockDim.x*gridDim.x)
11: end for
12: <GPU kernel End>
```

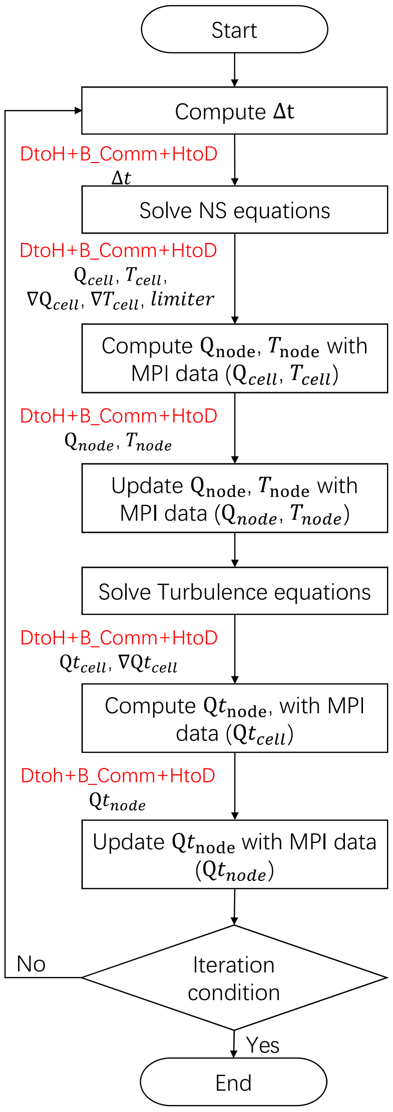
Algorithm 11 Local pressure comparing by cell loop (LPC-C)

```
1: <GPU kernel Begin>
2: threadID←threadIdx.x+blockIdx.x*blockDim.x
3: for cellID = threadID to nTotalCell-1 do
4:   numCell←numCellCell[cellID]
5:   cellStart←offsetCellCell[cellID]
6:   for cellInCellID = 0 to numCell-1 do
7:     cellCellID←cellCell[cellStart+cellInCellID]
8:     setCompMin(pMin[cellID], pressure[cellCellID])
9:     setCompMax(pMax[cellID], pressure[cellCellID])
10:  end for
11:  setAdd(cellID, blockDim.x*gridDim.x)
12: end for
13: <GPU kernel End>
```

4.2. Hybrid MPI and CUDA framework

A basic hybrid MPI-CUDA framework is established. Specifically, interface data transfer synchronously from Device to Host (DtoH). Then, by blocking MPI communication, interface data in host memory is communicated between different zones. Finally, interface data transfer synchronously from Host to Device (HtoD). Data transfer cannot overlap with data communication. Fig. 5 shows that five pairs of data transfer between host and device (HtoD and DtoH) in iterations, corresponding to time step (Δt), NS equations-related data (Q_{cell}, T_{cell}, ∇Q_{cell}, ∇T_{cell}, and *limiter*), NS interpolation-related data (Q_{nodes} and T_{nodes}), turbulence equation-related data (Qt_{cell}, ∇Qt_{cell}), turbulence interpolation-related data (Qt_{nodes}). The physical meaning of those data are described in Section 2. Data transfer exists on the default CUDA stream, which is synchronous with GPU computing.

After DtoH, interface data exchange between zones by blocking MPI communication (B_Comm). Data exchange is necessary for computing procedures. To illustrate, the time step (Δt) should be uniform in all zones. Some cell-centered data including Q_{cell}, T_{cell}, *limiter*, and Qt_{cell} should be updated on interfaces.



Start
Compute Δt
DtoH+B_Comm+HtoD
Δt
Solve NS equations
DtoH+B_Comm+HtoD
Q_cell, T_cell, ∇Q_cell, ∇T_cell, limiter
Compute Q_node, T_node with MPI data (Q_cell, T_cell)
DtoH+B_Comm+HtoD
Q_node, T_node
Update Q_node, T_node with MPI data (Q_node, T_node)
Solve Turbulence equations
DtoH+B_Comm+HtoD
Qt_cell, ∇Qt_cell
Compute Qt_node, with MPI data (Qt_cell)
Dtoh+B_Comm+HtoD
Qt_node
Update Qt_node with MPI data (Qt_node)
Iteration condition
No
Yes
End

Fig. 5. Basic MPI-CUDA framework.

Interpolation-related data including Q_{nodes}, T_{nodes}, and Qt_{nodes} exchange for gradient calculation. ∇Q_{cell}, ∇T_{cell}, and ∇Qt_{cell} exchange for computing with gradient on the interface. Those data exchange is blocked, which is also synchronous with GPU computing.

CUDA-aware MPI support is based on GPUDirect techniques supported by Nvidia, which can accelerate data exchange in the multi-GPU system. CUDA-aware MPI support technique can be directly used on the hybrid MPI-CUDA framework. PtoP data transfer between GPUs is observed in the same computing node. However, given H2D and D2H still exist on different computing nodes, GPUDirect RDMA does not work in the research. Only the first GPUDirect technology [35] works, by which the host memory copies and the network transfers can be pipe-lined.

4.3. Packing and unpacking interface data on GPU

Before data communication between zones, data packing should be performed to divide interface data into groups for neighbor zones. Similarly, after data communication, data unpacking is performed to rearrange groups of data received from neighbor zones on interfaces. Many threads loading and storing data on the GPU may accelerate data packing and unpacking.

Algo. 12 describes data packing on the GPU. Two arrays (*dataSend* and *dataIF*) are used to store packing and interface data, respectively (line 1). Then, a loop of neighbor zones is performed (Line 2). For each neighbor zone, the number of adjacent faces (numNgbFace), the start position (startSend) in the packing data (*dataSend*), and the start position (startFace) on the interface are determined (Line 3–5). A loop on adjacent faces (Line 6–8) is ported to the GPU. The packing data (*dataSend*) is set by interface data (*dataIF*) (line 13), by face reflection (line 11, 12) from a neighbor (faceID from 0 to numNgbFace-1) to the whole interface (sendID from 0 to nInterfaceTotal-1). Data unpacking described in Algo. 13 is the reverse of packing, where the receiving data (*dataRecv*) is set by the interface data (*dataIF*).

Algorithm 12 Pack MPI data on GPU

```
1:  Get dataSend and dataIF by data name
2:  for ngbZoneID = 0 to numNgbZone do
3:    startFace ← startFaceForSend[ngbZoneID]
4:    startSend ← startDataSend[ngbZoneID]
5:    numNgbFace ← nIFaceOfNgbZone[ngbZoneID]
6:    <GPU kernel Begin>
7:    threadID ← threadIdx.x+blockIdx.x*blockDim.x
8:    for faceID = threadID to numNgbFace do
9:      sendID ← faceForSend[startFace+faceID]
10:     for eqnID= 0 to numEqn do
11:       ngbZoneFaceID ← startSend+
          eqnID*numNgbFace+faceID
12:       interfaceID ← eqnID*nInterfaceTotal+sendID
13:       dataSend[ngbZoneFaceID] ← dataIF[interfaceID]
14:     end for
15:     setAdd(faceID, blockDim.x*gridDim.x)
16:   end for
17:   <GPU kernel End>
18: end for
```

Algorithm 13 Unpack MPI data on GPU

```
1:  Get dataSend and dataIF by data name
2:  for ngbZoneID = 0 to numNgbZone do
3:    startFace ← startFaceForRecv[ngbZoneID]
4:    startRecv ← startDataRecv[ngbZoneID]
5:    numNgbFace ← nIFaceOfNgbZone[ngbZoneID]
6:    <GPU kernel Begin>
7:    threadID ← threadIdx.x+blockIdx.x*blockDim.x
8:    for faceID = threadID to numNgbFace do
9:      recvID ← faceForRecv[startFace+faceID]
10:     for eqnID= 0 to numEqn do
11:       ngbZoneFaceID ← startRecv+
          eqnID*nIFaceOfNgbZone+faceID
12:       interfaceID ← eqnID*nInterfaceTotal+recvID
13:       dataIF[interfaceID] ← dataRecv[ngbZoneFaceID]
14:     end for
15:     setAdd(faceID, blockDim.x*gridDim.x)
16:   end for
17:   <GPU kernel End>
18: end for
```

5. Computing environment

5.1. Computing platform

In the research, numerical simulations are performed on a High Performance Computing (HPC) multi-GPU system. The HPC system contains 50 computing nodes. Each computing node owns 4 Nvidia Tesla V100 GPUs (16 GB device memory, bandwidth 900 GB/s), 2 Intel Xeon Gold 6132 CPUs (2.60 GHz, 14 CPU cores, bandwidth 124.8 GB/s each), and 256 GB of host memory (DDR4, 2666 MHz, 16 DIMMS of 16 GB). The Mellanox EDR interconnect fabric is used to connect all computing nodes by fat tree layout. NVLink 2.0 (throughput 150 GB/s) is used to connect GPUs, and PCIE3.0 (throughput 15 GB/s) is used to connect CPUs and GPUs in one computing node. The release version of the operating system is Centos 7.6 (kernel version 3.10.0-957.el7.x86_64). GCC 4.8.5 and NVCC (CUDA 10.0) are used for compiling host code and CUDA kernels, respectively. Open MPI 4.0.0 is applied for the exchange of data between processes.

Table 2
Five different meshes for M6 (million).

Meshes	Mesh 1	Mesh 2	Mesh 3	Mesh 4	Mesh 5
Cells	1.05	2.26	3.97	6.30	8.78
Faces	2.32	5.02	8.83	14.07	20.68

Table 3
Three different meshes for CHN-T1 (million).

Meshes	Mesh 6	Mesh 7	Mesh 8
Cells	17.37	35.20	102.59
Faces	39.40	70.59	205.56

5.2. Test cases

A series of numerical simulations are performed on two test cases. Firstly, external flow over an ONERA M6 wing is simulated with angle of attach 3.06 rad, side slip of angle 0 deg, and Mach number of 0.84. Five different mesh scales described in Table 2 are used to discuss the optimization effects on single-GPU.

Secondly, a transport airplane called CHN-T1 [36] in flight is simulated with angle of attach -1 rad, side slip of angle 0 deg, and Mach number of 0.78. Three different meshes described in Table 3 are applied to investigate the performance of the hybrid MPI-CUDA parallel framework on multi-GPU simulations.

5.3. Metrics of performance and scalability

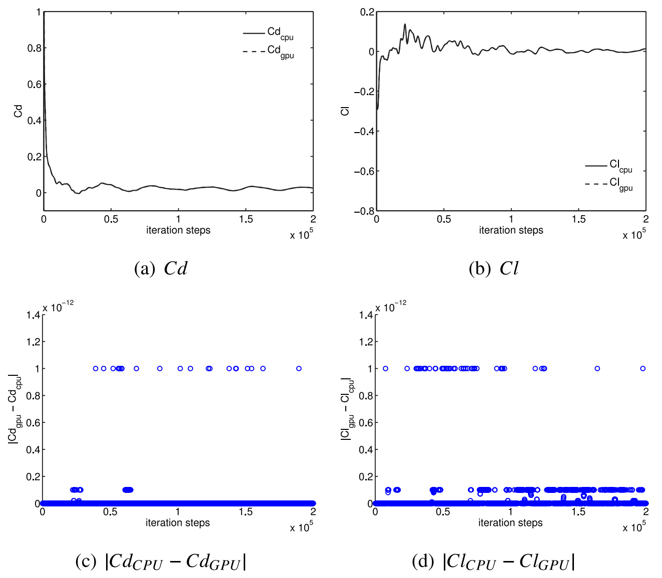
In the research, all floating-point operations are double precision (FP64).

The effects of optimization strategies on either kernel or application level can be evaluated by executing time. Specifically, on the kernel level, the executing time is recorded in the first 100 iteration steps on both the GPU and the CPU. The executing time of the application is recorded between 200 and 300 iteration steps. The variation of the overall executing time of Mesh 2 is evaluated by repeating ten measurements. The mean squared error is 0.0042, with an average of 17.14 on the GPU. On the CPU, the mean squared error is 0.42, with an average of 238.92.

Parallel efficiency is used to evaluate the scalability of multi-GPU computing. In the strong scaling test, the overall mesh scale is fixed so that strong parallel efficiency p_s is computed by $p_s = (t_b \cdot N_b)/(t_e \cdot N_e)$. In the weak scaling test, weak parallel efficiency p_w is computed by $p_w = t_b/t_e$ corresponding to a fixed partitioned mesh scale on each GPU. Executing time t_b and t_e are wall time consumed by baseline and evaluation applications, respectively. The numbers N_b and N_e are the number of MPI processes in baseline and evaluation applications.

6. Validation

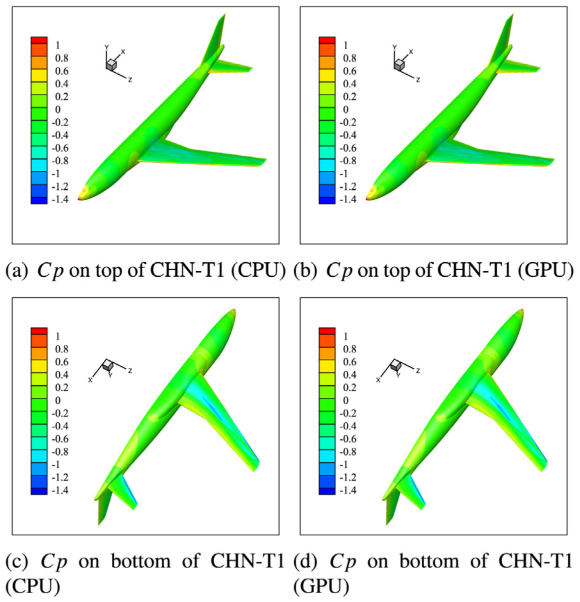
A simulation of CHN-T1 (Mesh 6) is performed using 72 GPUs, to validate the hybrid MPI-CUDA paralleled simulation on multi-GPU. After a series of optimizations, simulation results are compared with those using 72 CPUs. The comparison of drag coefficient *Cd* and lift coefficient *Cl* over 2,00,000 iteration steps in



(a) Cd (b) Cl

(c) $|Cd_{CPU} - Cd_{GPU}|$ (d) $|Cl_{CPU} - Cl_{GPU}|$

Fig. 6. Comparison of aerodynamics between CPU and GPU.



(a) Cp on top of CHN-T1 (CPU) (b) Cp on top of CHN-T1 (GPU)

(c) Cp on bottom of CHN-T1 (CPU) (d) Cp on bottom of CHN-T1 (GPU)

Fig. 7. Comparison of aerodynamics between CPU and GPU.

Fig. 6(a) and (b) indicates that the variations of aerodynamics with iterations are the same on both GPUs and CPUs. The difference described by $|Cd_{CPU} - Cd_{GPU}|$ and $|Cl_{CPU} - Cl_{GPU}|$ is shown in Fig. 6(c) and (d). The order of difference is less than 10^{-12},

Table 4
Time (unit: ms), Bandwidth (BW, unit: GB/s) and Sustained performance (SP, unit: GFLOP/s) on Mesh 3.

No.	FLOP/Byte	CPU			GPU		
		Time	BW	SP	Time	BW	SP
K1	0.09	8.30	49.69	4.42	1.63	252.49	22.44
K2	0.08	46.34	23.13	1.78	88.33	12.13	0.93
K3	0.52	195.65	7.11	3.68	9.40	148.08	76.51
K4	0.21	17.18	25.60	5.33	2.60	169.31	35.27
K5	0.18	73.35	15.18	2.72	6.13	181.75	32.56
K6	0.05	9.68	49.23	2.37	1.81	263.54	12.67
K7	0	8.43	46.26	0.00	1.41	276.10	0.00
K8	0.04	5.80	36.32	1.58	1.17	180.69	7.86
K9	0.37	28.40	21.57	7.92	1.63	374.69	137.67
K10	0.11	8.23	42.32	4.45	0.77	450.95	47.47

compared with the order of aerodynamics. Furthermore, the comparison of the pressure coefficient Cp in Fig. 7 indicates that the pressure distribution is the same. Finally, both the comparison of aerodynamic coefficients and the comparison of the pressure coefficient validate GPU simulation.

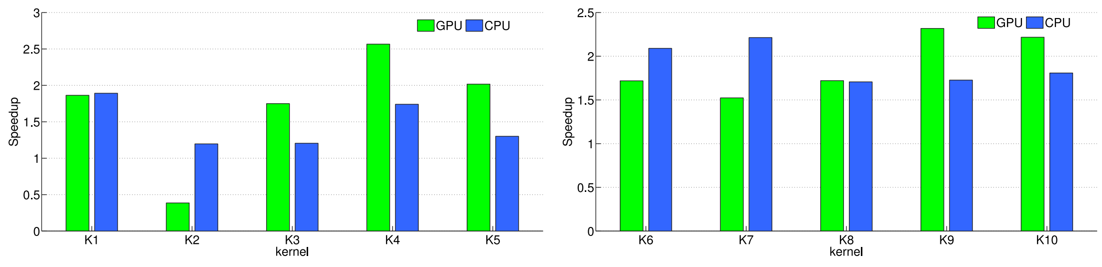
7. Effects of optimizations

7.1. Cell and face renumbering

In order to evaluate the effects of cell and face renumbering on executing time, the relative speedup compared to kernels without renumbering is shown in Fig. 8. Renumbering affects performance on the GPU and the CPU in a global way. Specifically, on the GPU, renumbering achieved 1.4–1.5× speedup in most kernels, except for data interpolation (K2). On the CPU, renumbering enhanced all kernels' performance. Finally, on both the GPU and the CPU, renumbering leads to 1.05–1.63× speedup on the whole application, shown in Fig. 11.

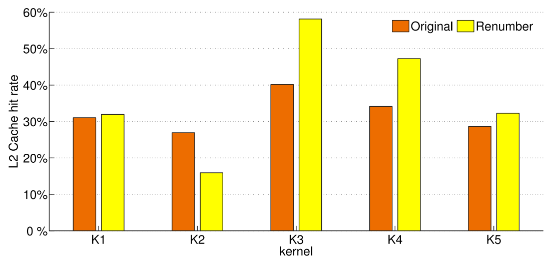
After renumbering, both memory bandwidth and sustained performance on the GPU and the CPU are described in Table 4. All kernels' computing performance is far from the theoretical peak values on the CPU (582.4 GFLOPS) and GPU (7.8 TFLOPS). All kernels' computing performance is memory-bound, compared with the theoretical FLOP byte ratio of the GPU (8.87 FLOP/byte) and the CPU (2.34 FLOP/byte). Furthermore, owing to indirect memory access, all kernels' memory bandwidth cannot reach the theoretical peak memory bandwidth of the GPU (900 GB/s) and the CPU (249.6 GB/s). Most kernels' memory bandwidth on the GPU is much larger than those on the CPU. Thus, the GPU plays a crucial in the computing performance of the application.

Effects of renumbering on memory utilization are investigated on the GPU. From L1 and L2 cache hit rate in Figs. 9 and 10, it indicates that both L1 and L2 cache hit rate are enhanced by arranging data distribution in global memory, leads to performance increase. For data interpolation (K2), due to a 12% increase in L1 cache hit rate and a 10% decrease in L2 cache hit rate,



(a) Kernel 1∼5 (b) Kernel 6∼10

Fig. 8. Kernels' speedup compared to original order on the GPU and the CPU.


(a) Kernel 1∼5

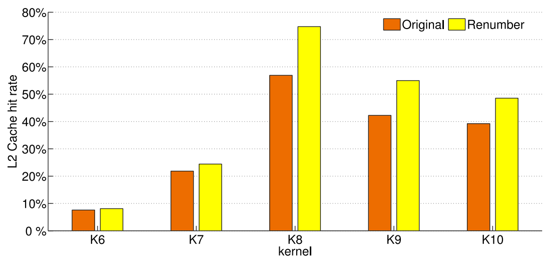
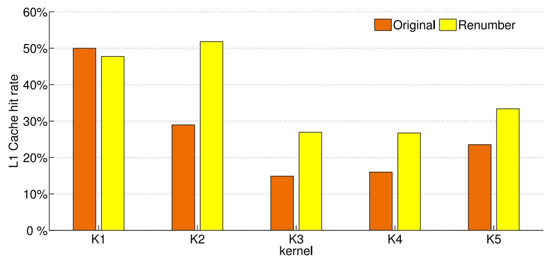

(b) Kernel 6∼10

Fig. 9. L2 cache hit rate of kernels with and without renumbering on GPU.


(a) Kernel 1∼5

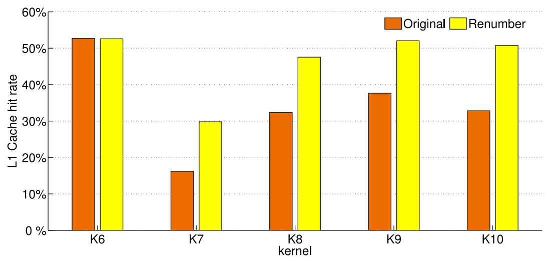

(b) Kernel 6∼10

Fig. 10. L1 cache hit rate of kernels with and without renumbering on GPU.

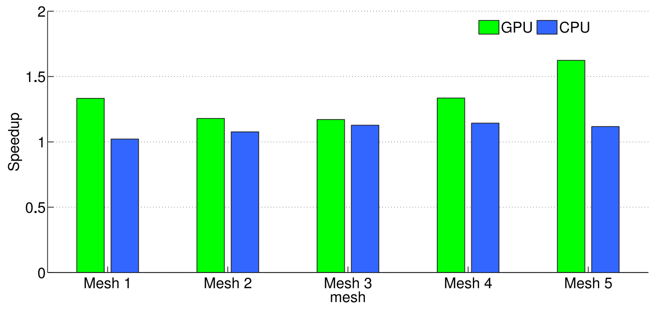


Fig. 11. The whole application's speedup compared to original order on GPU and CPU.

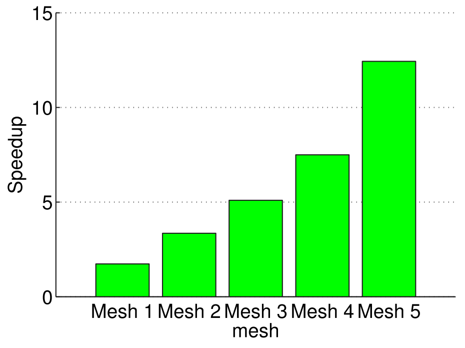


Fig. 12. Flux summation's speedup compared to graph coloring method on GPU.

there is a 16.9K decrease in memory write transactions and a 21.0K increase in memory read transactions at L2 cache. Due to much more time consumed in memory write transactions, the performance of the kernel K2 decreases.

7.2. Comparison between atomic operations and graph coloring method

On the GPU, the race condition exists in most kernels, as shown in Table 1. The performance of atomic operations and graph coloring is compared in flux summation (K2) to solve the race condition efficiently.

Fig. 12 shows that the atomic operation beats graph coloring on all mesh scales. With the increase of mesh scale, speedup compared to graph coloring increases. By re-grouping in the graph coloring method, indirect data access is aggravated, which leads to worse performance. Thus, atomic operations are applied in all kernels.

7.3. Nested loops split

Due to the widespread use of nested loops, memory throttle may be amplified by frequent data access. A large kernel may be split according to loop dimensions for efficient nested-loop computing.

Table 5
Comparison of nested loops split in flux summation (K6) on Mesh 3.

Memory utilization	Whole loop	Split loop
Time (ms)	1.81	1.69
BW (GB/s)	263.54	324.8
SP (GFLOP/s)	12.67	13.53
L1 cache hit rate	52.29%	42.86%
L2 cache hit rate	8.10%	11.49%
Memory read transactions (L2)	59464K	68219K
Memory write transactions (L2)	46508K	46798K

Fig. 13 shows the speedup compared to the whole kernel on both the GPU and the CPU. On the GPU, 1.05–1.1× speedup is obtained from Mesh 3 to Mesh 5. On the CPU, nested-loop splitting reduces computing performance.

In Table 5, the memory utilization on the GPU indicates a 9.43% decrease in the L1 cache hit rate and a 3.4% increase in the L2 cache hit rate. Despite 14.7% higher in memory read transactions (L2 cache), memory write transactions related to atomic operations remain similar. Nested loops split eased memory throttle due to atomic operations, which leads to 23.24% higher memory bandwidth.

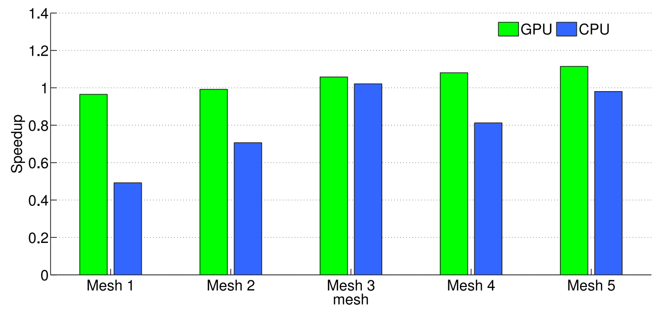


Fig. 13. Flux summation's speedup compared to original nested loops on GPU and CPU.

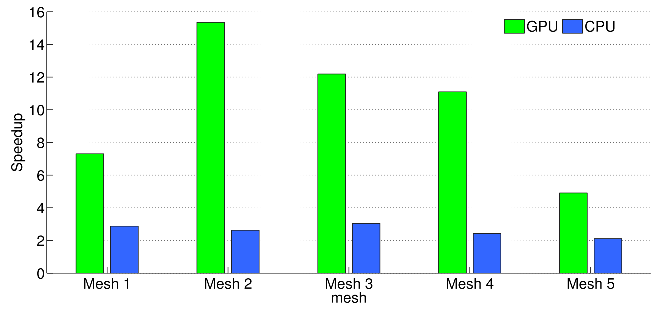


Fig. 14. Data interpolation's speedup compared to the face loop on GPU and CPU.

Table 6
Comparison of loop modes in data interpolation (K2) on Mesh 3.

Memory utilization	Face loop	Cell loop
Time (ms)	88.33	7.16
BW (GB/s)	12.12	314.68
SP (GFLOP/s)	0.93	15.35
L1 cache hit rate	51.80%	73.73%
L2 cache hit rate	15.92%	8.09%
Memory read transactions (L2)	446492K	177907K
Memory write transactions (L2)	371102K	53233K

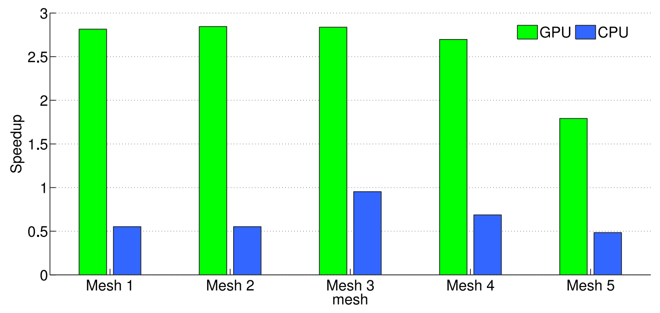


Fig. 15. Pressure computing's speedup compared to the face loop.

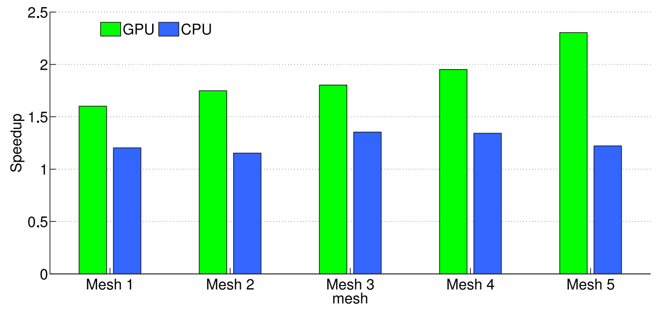


Fig. 16. The whole application's speedup compared to original without optimizations on GPU and CPU.

Table 7
Comparison of loop modes in pressure computing (K8) on Mesh 3.

Memory utilization	Face loop	Cell loop
Time (ms)	1.17	0.43
BW (GB/s)	180.69	469.17
SP (GFLOP/s)	7.86	42.65
L1 cache hit rate	47.54%	63.11%
L2 cache hit rate	74.73%	71.47%
Memory read transactions (L2)	37313K	16057K
Memory write transactions (L2)	21715K	8567K

7.4. Loop modes adjustment

The face loop is applied in most kernels. However, face data does not exist in some kernels, such as data interpolation (K2) and local pressure computing (K8). The face loop aggravates overheads induced by indirect memory access. Furthermore, after cell and face renumbering, the reduced L2 cache hit rate makes data interpolation consume too much time on the GPU. Hence, the cell loop (Algo. 8) instead of the face loop (Algo. 7) is applied in K2, to improve data locality.

Fig. 14 shows the speedup compared to the face loop on both the GPU and the CPU. 4.5–15.6× speedup is achieved on the GPU, while 2.1–3.2× speedup on the CPU.

The memory utilization on the GPU described in Table 6 shows that a 21.9% increase in L1 cache hit rate and a 6.8% decrease in L2 cache hit rate lead to a significant reduction in both memory read and write transactions (L2). The cell loop makes sustained performance 16.5 times higher and memory bandwidth 29.6 times higher.

Like data interpolation, local pressure computing (K8) only compares the cell-centered array *pressure* by the face loop. Thus, cell loop (Algo. 11) can ease indirect data access.

Fig. 15 shows that the cell loop reduced executing time remarkably on the GPU, achieving speedups of 1.72–2.74 compared to the face loop. On the contrary, the cell loop consumed more executing time on the CPU, corresponding to the speedup below 1.

In Table 7, the memory utilization on the GPU shows that a 15.6% increase in L1 cache hit rate and a 3.2% decrease in L2 cache hit rate result in over 50% reduction in both memory read and write transactions (L2). By the cell loop, 16.5 times higher sustained performance and 29.6 times higher memory bandwidth are achieved.

Both data interpolation and local pressure comparing show that the cell loop is more convenient for kernels without face data on the GPU where data locality is improved.

7.5. Overall performance on a computing node

A series of optimization strategies are performed on both the CUDA code and the host code, including cell & face renumbering, nested loops split, and loop modes adjustment. Fig. 16 shows that 1.52–2.27× speedup compared to simulations without optimizations is achieved on the GPU, while 1.15–1.32× speedup on the CPU. Contributions of optimization strategies are different on the GPU. Fig. 17 indicates that renumbering contributes to a 14.8%–38.2% increase in overall performance. Loop modes adjustment achieves a 14.2%–19.6% overall performance increase from Mesh 2 to Mesh 5. Nested loops split gives a 4.9%–10.9% performance boost in the whole application.

In Fig. 18, the overall performance comparison shows that a single GPU achieves 11.9–14.8× speedup compared to 28 CPU

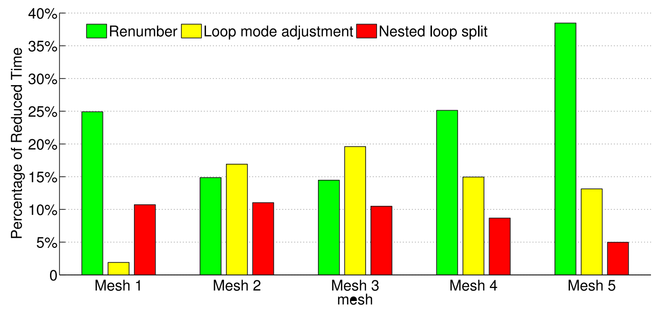


Fig. 17. Contributions of optimizations on GPU.

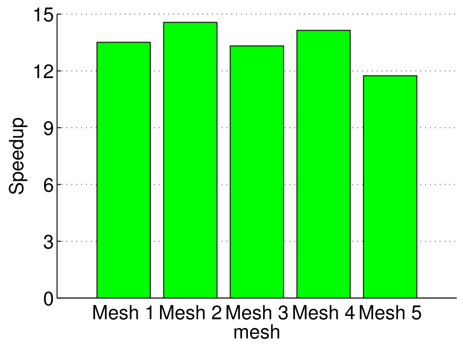


Fig. 18. A GPU's Overall performance speedup compared to 28 CPU cores.

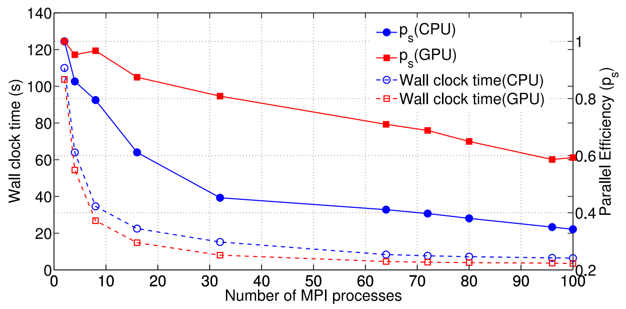


Fig. 19. Overall performance and parallel efficiency comparison between data packing/unpacking on the GPU and the CPU.

cores on different mesh scales. In fact, from analysis in Section 7.1, the peak theoretical memory bandwidth is not reached on both the GPU and the CPU. At the current stage, the speedup compared to CPU computing is reasonable, according to some studies [37].

8. Multi-GPU computing

8.1. Data packing and unpacking on the GPU

For making use of many threads loading and storing data on the GPU, interface data is packed and unpacked on the GPU. The effects of GPU packing and unpacking data on overall performance are investigated on Mesh 6.

Fig. 19 shows that data packing and unpacking on the GPU consumes 50% less wall clock time than on the CPU.

Parallel efficiency (p_s) at the baseline of 2 GPUs shows that data packing and unpacking on the GPU owns twice higher parallel efficiency than on the CPU. Given performance and scalability,

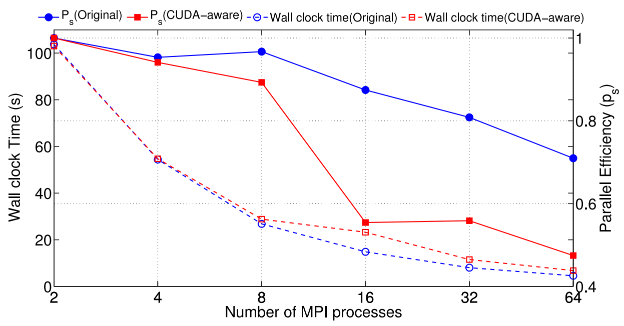


Fig. 20. Wall clock time and parallel efficiency on the hybrid MPI-CUDA parallel framework.

Table 8
Effects of mesh scale on parallel efficiency (p_s) in strong scaling test.

No. of MPI processes	Mesh 6		Mesh 7		Mesh 8	
	Time (s)	p_s	Time (s)	p_s	Time (s)	p_s
64	4.57	1	8.16	1	25.47	1
96	3.68	82.7%	5.93	91.8%	16.61	102.2%
160	2.78	65.8%	3.94	82.8%	9.62	105.9%

data packing and unpacking on the GPU is more efficient for loading and storing data.

8.2. The hybrid MPI-CUDA parallel framework

We evaluate wall clock time and scalability of the hybrid MPI-CUDA parallel framework on multi-GPU computing by simulations on Mesh 6 from 2 GPUs to 64 GPUs.

Fig. 20 shows that the CUDA-aware MPI support consumes the highest wall time since 8 GPUs, which takes more than 40% time over the original framework. Compared with some studies [29,30], the CUDA-aware MPI support does not work well. Although host memory copies and network transfers are pipelined by GPUDirect techniques, many synchronizations stem from frequent MPI communication required by the turbulence model and gradient calculation. Besides wall clock time, the parallel efficiency (p_s) of the CUDA-aware MPI support is also lower than the original framework.

Given wall clock time and parallel efficiency, the CUDA-aware MPI support has the worst performance. The original framework is used in multi-GPU computing.

8.3. Strong scaling test

We evaluate the strong scalability of the hybrid MPI-CUDA parallel framework. Due to the limit of GPU global memory, we use 2 GPUs, 4 GPUs, and 12 GPUs for baseline simulations corresponding to Mesh 6, Mesh 7, and Mesh 8. Fig. 21 shows that on Mesh 6 (the coarse mesh), at the baseline of 2 GPUs, a parallel efficiency using 200 GPUs is higher than 40%. On Mesh 7 (the moderate mesh), at the baseline of 4 GPUs, a parallel efficiency using 160 GPUs can reach 75%. On Mesh 8 (the fine mesh), after 64 GPUs, the parallel efficiency at the baseline of 12 GPUs is still above 1. On Mesh 6 and 7, the parallel efficiency descends with the increasing number of MPI processes due to data transfers and communication.

We also evaluate the effects of mesh scale on scalability. At the baseline of 64 GPUs, the parallel efficiency (p_s) is compared between simulations using 96 and 160 GPUs on Mesh 6, Mesh 7, and Mesh 8. Table 8 shows that the parallel efficiency (p_s) using 96 GPUs increases from 82.7% to 102.2%, with the mesh scale increasing from Mesh 6 to Mesh 8. With mesh scale increasing,

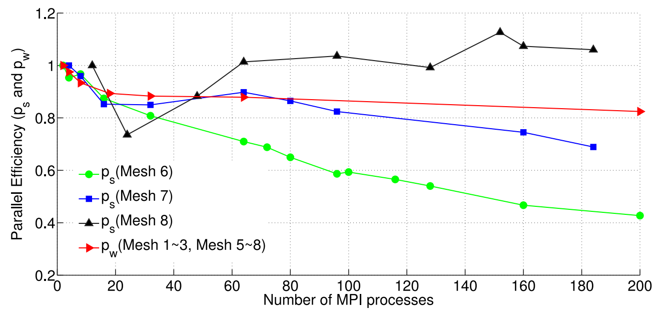


Fig. 21. Strong and weak scaling test.

computing accounts for a higher portion of executing time, resulting in better parallel efficiency, which is more evident using more GPUs.

8.4. Weak scaling test

We perform a weak scaling test on the hybrid CUDA-MPI parallel framework, with about 500,000 cells per GPU, corresponding to Mesh 1 (2 GPUs), Mesh 2 (4 GPUs), Mesh 3 (8 GPUs), Mesh 5 (18 GPUs), Mesh 6 (32 GPUs), Mesh 7 (64 GPUs), and Mesh 8 (200 GPUs). We use 2 GPUs on Mesh 1 as the baseline. Fig. 21 shows that we obtain good parallel efficiency higher than 80% in those cases. Parallel efficiency (p_w) using 200 GPUs is almost 85%. The loss of parallel efficiency stems from data exchange between GPUs.

9. Conclusion

In this paper, an unstructured mesh-based finite volume CFD solver for compressible flow is ported on Nvidia GPU V100. Some strategies are applied for optimizing indirect data access. Both packing and unpacking data for communication are performed on the GPU. A hybrid MPI-CUDA parallel framework with data packing and unpacking on the GPU is established for multi-GPU computing. Both strong and weak scaling tests are performed on the hybrid MPI-CUDA framework.

It is found that cell and face renumbering optimizes the global data location, leading to most kernels' performance increase by 40%–60%. For resolving data dependence problems, atomic operations offer more than double the performance of graph coloring. Nested-loop splitting eased non-coalescing data access, resulting in some kernels' executing time decreased by 10% on the large mesh. The cell loop improves data locality for those kernels without face data, which brings more than five times the performance of the face loop interpolating data. After those optimizations, overall performance on a GPU is enhanced by around 50%. For whole applications on different mesh scales, an average speedup of 13.45 on a single GPU (Nvidia Tesla V100) is achieved, compared to 28 CPU cores (Intel Xeon Gold 6132).

Multi-GPU simulations indicate that data packing and unpacking on the GPU is almost double the performance and the parallel efficiency of those on the CPU. The hybrid MPI-CUDA framework with CUDA-aware MPI support is weakened by extra synchronizations, leading to worse performance and efficiency. The strong scaling test shows that a parallel efficiency of 40% achieves on 200 GPUs, at the baseline of 2 GPUs. A good weak scaling parallel efficiency higher than 80% is achieved by 200 GPUs to 2 GPUs with 500 thousand cells per GPU.

10. Future work

In the future, mixed precision computing will be studied on the GPU. Effects of SIMD vectorization will be studied on different types of CPUs.

CRediT authorship contribution statement

Xi Zhang: Writing, Profiling, Program development. **Xiaohu Guo:** Participated in the discussion of ideas remotely. **Yue Weng:** Helped in this manuscript writing, Profiling and the development of algorithms. **Xianwei Zhang:** Helped profiling and performed the analyze of the program bottleneck. **Yutong Lu:** Participated in the discussion and provided platform and computing power support. **Zhong Zhao:** Helped in discuss.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Zhang Xi reports financial support was provided by China Aerodynamics Research and Development Center. Zhang Xi reports financial support was provided by Sun Yat-Sen University.

Data availability

Data will be made available on request.

Acknowledgment

The research is funded by National Numerical Wind tunnel project, China NNW2019ZT6-B18 and Guangdong Introducing Innovative and Entrepreneurial Teams, China under Grant No. 2016ZT06D211. We thank three anonymous reviewers for their insightful suggestions. We express gratitude to Dr. Lixiang Luo at IBM Research for his suggestions on algorithm implementation. We would like to thank Dr. Hanfeng Gu at National Supercomputer Center in Wuxi for his help in cell and face renumbering. We are grateful to Dr. Zhiyong Bai in China Aerodynamic Research and Development Center for discussing CUDA-aware MPI support technique. The first author thanks the staff at the HPC department in NSCC-GZ. Special thanks to Mr. Xu Sun for developing many crucial CUDA kernels. The first author appreciates the continuous encouragement and assistance of his wife, Mrs. Xueyu Chen. All authors approved the version of the manuscript to be published.

References

[1] M.A. Park, A. Loseille, J. Krakos, T.R. Michal, J.J. Alonso, Unstructured grid adaptation: Status, potential impacts, and recommended investments towards CFD 2030, in: 2016 the 46th AIAA Fluid Dynamics Conference, 2016.

[2] L. Cirrottola, M. Ricchiuto, A. Froehly, B. Re, A. Guardone, G. Quaranta, Adaptive deformation of 3D unstructured meshes with curved body fitted boundaries with application to unsteady compressible flows, J. Comput. Phys. 433 (2021) 110177.

[3] B. Diskin, W.K. Anderson, M.J. Pandya, C.L. Rumsey, H. Nishikawa, Grid convergence for three dimensional benchmark turbulent flows, in: 2018 AIAA Aerospace Sciences Meeting, 2018.

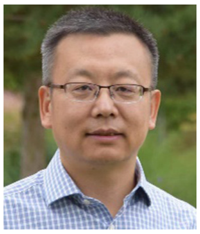
[4] T.D. Economon, F. Palacios, S.R. Copeland, T.W. Lukaczyk, J.J. Alonso, SU2: An open-source suite for multiphysics simulation and design, AIAA J. 54 (3) (2016) 828–846.

[5] H.G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques, Comput. Phys. 12 (6) (1998) 620–631.

[6] C. Pain, A. Umpleby, C. de Oliveira, A. Goddard, Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations, Comput. Methods Appl. Mech. Engrg. 190 (29) (2001) 3771–3796.

[7] X. He, Z. Zhao, R. Ma, N. Wang, L. Zhang, Validation of HyperFLOW in subsonic and transonic flow, Acta Aerodyn. Sinica 34 (2) (2016) 267–275.
[8] J. Bakosi, R. Bird, F. Gonzalez, C. Junghans, W. Li, H. Luo, A. Pandare, J. Waltz, Asynchronous distributed-memory task-parallel algorithm for compressible flows on unstructured 3D Eulerian grids, Adv. Eng. Softw. 160 (2021) 102962.
[9] T.M. Aamodt, W. Fung, T.G. Rogers, General-purpose graphics processor architectures, Synthesis Lect. Comput. Archit. 13 (JUN.) (2018) 1–140.
[10] J. Xu, H. Fu, W. Luk, L. Gan, W. Shi, W. Xue, C. Yang, Y. Jiang, C. He, G. Yang, Optimizing finite volume method solvers on nvidia GPUs, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2790–2805.
[11] G. Rokos, G. Gorman, P.H.J. Kelly, A fast and scalable graph coloring algorithm for multi-core and many-core architectures, in: European Conference on Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 414–425.
[12] Professional CUDA C Programming, first ed., Wrox Press Ltd. GBR, 2014.
[13] A. Walden, E. Nielsen, B. Diskin, M. Zubair, A mixed precision multicolor point implicit solver for unstructured grids on GPUs, in: 2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms, IA3, 2019, pp. 23–30.
[14] A. Azad, M. Jacquelin, A. Buluç, E.G. Ng, The reverse Cuthill-McKee algorithm in distributed-memory, in: 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2017, pp. 22–31.
[15] A. Corrigan, F.F. Camelli, R. L?Hner, J. Wallin, Running unstructured grid-based CFD solvers on modern graphics hardware, Internat. J. Numer. Methods Fluids 66 (2) (2011) 221–229.
[16] A. Corrigan, J. Dahm, Unstructured grid numbering schemes for GPU coalescing requirements, in: GPU Technology Conference 2012, URL https://on-demand.gputechconf.com/gtc/2012/presentations/S0031-Unstructured-Grid-Numbering-Schemes-for-GPU-Coalescing-Requirements.pdf.
[17] A. Lani, M.S. Yalim, S. Poedts, A GPU-enabled finite volume solver for global magnetospheric simulations on unstructured grids, Comput. Phys. Comm. 185 (10) (2014) 2538–2557.
[18] M. Garcia-Gasulla, F. Mantovani, M. Josep-Fabrego, B. Eguzkitza, G. Houzeaux, Runtime mechanisms to survive new HPC architectures: A use case in human respiratory simulations, Int. J. High Perform. Comput. Appl. 34 (1) (2020) 42–56.
[19] M. Fuhry, A. Giuliani, L. Krivodonova, Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws, Internat. J. Numer. Methods Fluids 76 (12) 982–1003.
[20] A. Giuliani, L. Krivodonova, Face coloring in unstructured CFD codes, Parallel Comput. 63 (2017) 17–37.
[21] A.A. Sulyok, G.D. Balogh, I.Z. Reguly, G.R. Mudalige, Locality optimized unstructured mesh algorithms on GPUs, J. Parallel Distrib. Comput. 134 (2019) 50–64.
[22] X. Zhang, X. Sun, X. Guo, Y. Du, Y. Lu, Y. Liu, Re-evaluation of atomic operations and graph coloring for unstructured finite volume GPU simulations, in: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD, 2020, pp. 297–304.
[23] C.P. Stone, A. Walden, M. Zubair, E. Nielsen, Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs, in: 2021 IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms, 2021, pp. 19–26.
[24] M. Giles, G. Mudalige, B. Spencer, C. Bertolli, I. Reguly, Designing OP2 for GPU architectures, J. Parallel Distrib. Comput. 73 (11) (2013) 1451–1460.
[25] A. Rahimi, A. Ghofrani, M.A. Lastras Montano, K.T. Cheng, L. Benini, R.K. Gupta, Energy-efficient GPGPU architectures via collaborative compilation and memristive memory-based computing, in: 2014 Proceedings of the 51st Annual Design Automation Conference, 2014, pp. 1–6.
[26] D. Jacobsen, J. Thibault, I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, in: 2010 the 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, 2010, pp. 1–16.
[27] H. Zolfaghari, B. Becsek, M.G. Nestola, W.B. Sawyer, R. Krause, D. Obrist, High-order accurate simulation of incompressible turbulent flows on many parallel GPUs of a hybrid-node supercomputer, Comput. Phys. Comm. 244 (2019) 132–142.
[28] P. Vincent, F. Witherden, B. Vermeire, J.S. Park, A. Iyer, Towards green aviation with python at petascale, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 1–11.
[29] J. Romero, J. Crabill, J. Watkins, F. Witherden, A. Jameson, ZEFR: A GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method, Comput. Phys. Comm. 250 (2020) 107169.
[30] E. Jourdan, Z.J. Wang, Efficient implementation of the FR/CPR method on GPU clusters for industrial large eddy simulation, in: AIAA AVIATION 2020 FORUM, 2020, pp. 1–19.
[31] G. Oyarzun, R. Borrell, A. Gorobets, A. Oliva, Portable implementation model for CFD simulations. Application to hybrid CPU/GPU supercomputers, Int. J. Comput. Fluid Dyn. 31 (9) (2017) 396–411.
[32] X. Álvarez-Farré, A. Gorobets, F.X. Trias, A hierarchical parallel implementation for heterogeneous computing. application to algebra-based CFD simulations on hybrid supercomputers, Comput. & Fluids 214 (2021) 104768.
[33] R. Borrell, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, O. Lehmkuhl, V. Mehta, H. Owen, M. Vázquez, G. Oyarzun, Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics, Future Gener. Comput. Syst. 107 (2020) 31–48.
[34] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.
[35] J. Kraus, An Introduction to CUDA-Aware MPI. URL https://developer.nvidia.com/blog/introduction-cuda-aware-mpi.
[36] Z. Zhong, M.B. Rong, H.B. Lei, B. Xca, B. Lza, An efficient large-scale mesh deformation method based on MPI/OpenMP hybrid parallel radial basis function interpolation, Chin. J. Aeronaut. 33 (5) (2020) 1392–1404.
[37] I. Alonso Asensio, A. Alvarez Laguna, M.H. Aissa, S. Poedts, N. Ozak, A. Lani, A GPU-enabled implicit finite volume solver for the ideal two-fluid plasma model on unstructured grids, Comput. Phys. Comm. 239 (2019) 16–32.

Xi Zhang is an engineer at National Supercomputing Center in Guangzhou, School of Computer Science and Engineering, Sun Yat-sen University. He obtained Ph.D. at City University Of London (Hydraulic Engineering, 2018) and Harbin Engineering University (Engineering Mechanics, 2012). He leads the research of *Computational Fluid Dynamics in a large-scale heterogeneous computing system* granted by the National Numerical Wind tunnel project. His research interests include High Performance Computing and Computational Fluid Dynamics.



Xiaohu Guo is a Principal Computational Scientist and leading Intelligent Computing and Application Algorithms team (ICAA) in the Hartree Centre, STFC Daresbury Laboratory. He has been leading the development of computational numerical methods using particle and unstructured mesh based methods for a range of application fields including CFD, Materials Science, and Medical image processing and reconstruction. His work has provided enabling technologies for a wide range of engineering and science applications spanning the brief of the UKRI EPSRC, STFC and NERC research councils. Dr. Guo is also member of technical program committee for international key HPC conferences including SC, ICCS, and ISC in the high performance computing field, and the specialist editor of Computer Physics Communications.

Yue Weng is a postgraduate student at School of Computer Science, Sun Yat-sen University. He received his bachelor's degree from South China University of Technology in 2020. His research is conducted in the areas of high-performance computing and heterogeneous computing. He is very interested in performance optimization and multi-task hybrid scheduling for GPU applications. In addition, he has also done some research and achievements on cross-modal problems in deep learning.

Xianwei Zhang is an Associate Professor at School of Computer Science and Engineering of Sun Yat-sen University. During 2017-2020, he worked in AMD Inc. (Research, RTG) on hardware and software designs for compute-optimized GPUs. Xianwei completed his Ph.D (2017) in the Computer Science Department at University of Pittsburgh, and obtained his Bachelor's (2011) degree from Northwestern Polytechnical University. Xianwei's research interests lie broadly in hardware and software co-designs to improve the performance and efficiency of computing systems, including computer architecture, compiling, and HPC.

Yutong Lu is the Director of the National Supercomputing Center in Guangzhou, and she is a professor at Sun Yat-sen University. Her extensive research work has spanned several generations of China's Tianhe and earlier supercomputers, including working as the deputy chief designer of Tianhe-2 Supercomputer. Lu has won several national and provincial awards for science and technology advancement and is currently leading a number of HPC and big data research projects. She has been a member of the Expert Committee of China's National Key R&D Program of HPC since 2016 and is also a member of the CCF Council and the deputy editor of IEEE Transactions on Parallel and Distributed Systems. In 2017, she was named ISC Fellow.

Zhong Zhao is an associate professor at China Aerodynamics Research and Development Center (CARDC), and graduated from the Mathematics Department of Xi'an Jiaotong University. He leads the development of open source CFD code PHengLEI, which is famous for its excellent parallel efficiency in China. His research interests include unstructured grid generation, CFD methods and software development, and parallel computation.