doi:10.11887/j.cn.202205003 http://journal.nudt.edu.cn

面向GPU的非结构网格有限体积计算流体力学的图染色方法优化*

张 曦[1],孙 旭[1],郭晓虎[2],杜云飞[1],卢宇彤[1],刘 杨[3]
(1. 中山大学 计算机学院(软件学院),广东 广州 510006;
2. 哈璀国家超算中心 达斯伯里实验室,英国 沃林顿 WA4 4AD;
3. 中国空气动力研究与发展中心,四川 绵阳 621000)

摘 要:采用图染色方法解决通量累加和局部最大压力计算引起的两种典型资源竞争问题,并通过共享内存的使用、体编号和面编号的重排、面数据的重排三种策略优化图染色方法。针对应用在空气动力学多种规模的三维网格,分别采用双精度和单精度操作数,在Nvidia Tesla V100和K80 GPU上,展开性能测试。结果表明:共享内存的优化效果不明显;体编号和面编号重排降低了图染色方法的计算性能;面数据重排可以有效地优化图染色方法;计算性能在V100上提高20%左右,在K80上提高15%左右。

关键词:非结构网格;有限体积;图形处理器;资源竞争;图染色
中图分类号:TN95 **文献标志码**:A **文章编号**:1001-2486(2022)05-024-11

Optimizations of graph coloring method for unstructured finite volume computational fluid dynamics on GPU

ZHANG Xi[1], *SUN Xu*[1], *GUO Xiaohu*[2], *DU Yunfei*[1], *LU Yutong*[1], *LIU Yang*[3]
(1. School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China;
2. STFC Daresbury Laboratory, Hartree Centre, Warrington WA4 4AD, UK;
3. China Aerodynamics Research and Development Center, Mianyang 621000, China)

Abstract: Graph coloring was used to address resource competition for the two typical computing procedures, including the flux summation and the calculation of local maximum pressure. There were three optimizations applied on graph coloring including shared memory, the reordering of volume and face indices, and the reordering of face variables. The 3D aerodynamics application with a series of mesh sizes was used in the performance test by double and single precision floating point operations on GPU Nvidia Tesla V100 and K80. The performance tests show that the shared memory is not obvious in performance. Furthermore, the reorder of volume and face indices reduces the performance of graph coloring. It is found that the reorder of face variables can increase performance remarkably. Specifically, the performance of graph coloring is increased by around 20% on V100 and 15% on K80.

Keywords: unstructured mesh; finite volume; graphic processing units; race condition; graph coloring

非结构网格有限体积方法在航空航天、海洋环境等许多科学计算领域得到了广泛的应用,形成了一些优秀的数值模拟软件,例如OpenFOAM[1]、FUN3D[2]、Fludity[3]、NNW-PHengLEI(风雷)[4]等。随着高性能计算技术的发展,科学计算进入"E级"时代,数值模拟软件的计算性能也不断提高。特别是通用图形加速器(general purpose graphics processing unit, GPGPU)的快速发展,计算能力强、功耗低的GPU在科学计算中起到了越来越重要的作用。如何充分利用GPU,提高非结构网格有限体积数值模拟的计算性能,已经成为高性能计算领域的重要课题之一。

GPU性能的发挥主要取决于硬件占用率和访存延迟。非结构网格的数据存储是不规则的,需要通过几何拓扑关系访问数据,例如通过面循环访问体变量,这将导致频繁的不规则内存访问,增大访存延迟。因此,如何利用GPU多级存储体系减小访存延迟是提高应用性能的关键。

几何拓扑依赖读写模式在多线程并行计算中,可能造成多个线程改变相同的全局内存或共享内存,导致错误的计算结果,即资源竞争问题。图染色[5]是解决资源竞争问题的一种重要方法。

* **收稿日期**:2020-11-09
基金项目:国家重点研发计划资助项目(2016YFB0200902);国家数值风洞工程资助项目(NNW2019ZT6-B18);广东省引进创新创业团队资助项目(2016ZT06DZ11)
作者简介:张曦(1985—),男,河北石家庄人,工程师,博士,E-mail:zhangx299@mail.sysu.edu.cn

该方法将发生冲突的面通过染色分成若干组,以保证每一组中的任何两个面都不会更改相同的体变量。多线程并行计算可以在每一组中正确地实现。

图染色方法在可移植性和计算性能方面具有优势。图染色方法通过常规的编程语言即可实现,几乎可以移植到任何一种硬件平台。一些学者在有限元、有限体积计算流体力学(computational fluid dynamics, CFD)的实际案例中发现图染色方法具有较好的性能[6-7]。FUN3D[8]和 SU2[9]也采用了图染色方法解决多线程计算中的资源竞争问题,但都集中于 CPU。本文讨论了图染色方法对于 GPU 计算性能的影响。

图染色方法在分组之后,每一组的计算仍然存在不规则访存的问题,需要进一步优化。Giuliani 等[10]针对简单的几何拓扑关系,提出了一种优化方法,但无法将其应用在未知量定义在体中心的计算。Sulyok 等[11]也提出了优化方法,但是只针对一种网格尺寸进行了性能分析。本研究的图染色方法可应用在多种规模的复杂三维非结构网格。

1　非结构网格有限体积 CFD 数学模型

本研究采用的非结构网格有限体积 CFD 程序是 NNW-PHengLEI(风雷)。该程序由中国空气动力研究与发展中心研发。

1.1　控制方程与离散

控制方程为积分形式的 NS(Navier-Stokes)方程,表示为

$$\frac{\partial}{\partial t}\int_{\Omega}\boldsymbol{Q}\mathrm{d}\Omega+\oint_{\partial\Omega}(\boldsymbol{F}_{\mathrm{c}}-\boldsymbol{F}_{\mathrm{v}})\mathrm{d}S=0 \tag{1}$$

其中:$\boldsymbol{F}_{\mathrm{c}}$ 和 $\boldsymbol{F}_{\mathrm{v}}$ 分别为对流通量和黏性通量;$\boldsymbol{Q}$ 包含 5 个未知数,可以表示为

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

式中,ρ 表示流体密度,u、v、w 分别表示速度在 x、y、z 三个方向的分量,e 表示内能。这些物理量定义在控制体 Ω 中心上。

有限体积方法将物理空间划分为许多小的控制体(volume),体的边界称为面(face)。相应地,NS 方程的离散形式可以表示为

$$\frac{\Delta\boldsymbol{Q}_{\mathrm{vol}}}{\Delta t}=-\frac{1}{\Omega_{\mathrm{vol}}}\left[\sum_{m=1}^{N_{\mathrm{F}}}(\boldsymbol{F}_{\mathrm{c}}-\boldsymbol{F}_{\mathrm{v}})_m\cdot S_m\right] \tag{3}$$

式中:Ω_{vol} 表示控制体的体积;控制体拥有 N_{F} 个面,每个面的面积为 S_m;对流通量 $\boldsymbol{F}_{\mathrm{c}}$ 采用 Roe 格式计算,并且采用限制器抑制激波产生的数值振荡;黏性通量 $\boldsymbol{F}_{\mathrm{v}}$ 采用中心差分格式计算;Splart Allmaras 方程湍流模式用来计算湍流的影响;显格式龙格库塔方法用来离散时间项;最终通过对流通量和黏性通量的残差来更新 $\boldsymbol{Q}$,保证计算的收敛。程序在一个时间步的流程如图 1 所示。

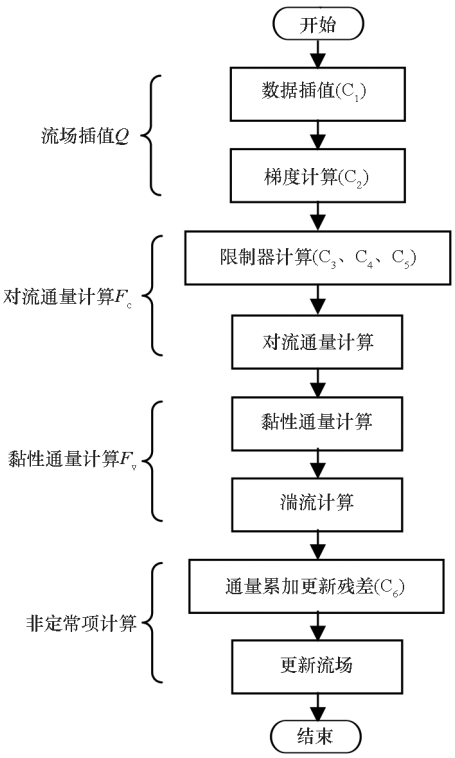


图 1　程序在一个时间步的流程

Fig. 1　Flow chart in one time step

1.2　网格拓扑和数据结构

非结构网格几何拓扑关系由体编号和面编号来表示。非结构网格通常采用面的编号顺序来存储体编号。例如将每个面对应的较小的体编号存储在 *owner* 中,将较大的体编号存储在 *neighbor* 中。如图 2 所示,编号为 O 和 P 的 2 个体($O<P$)都拥有编号为 f_2 的面,可以得到 $owner[f_2]=O$,$neighbor[f_2]=P$。

有限体积 CFD 程序中,变量 ρ、u、v、w、e 等存储在体中心,称为体数据;对流通量 $\boldsymbol{F}_{\mathrm{c}}$ 和黏性通量 $\boldsymbol{F}_{\mathrm{v}}$ 等变量则存储在面中心,称为面数据。体数据和面数据由各自的编号索引,并以结构体数组(structure of array,SOA)方式存储。

非结构网格的几何拓扑关系和数据存储方

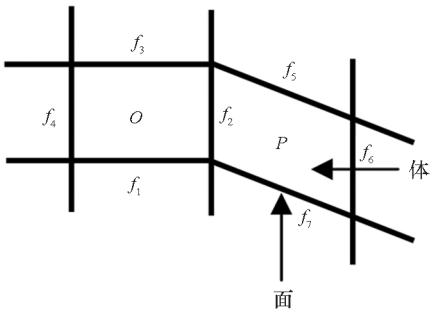


图 2 几何拓扑关系:体和面

Fig. 2 Geometry topology: volume cell and face

式使得体数据可以由体编号直接索引,也可以由面编号间接索引。例如表示残差的体变量 *res*,其编号 O 的数据可以表示为 $res[O]$ 或者 $res[owner[f_2]]$。

1.3 资源竞争

在 NNW-PHengLEI(风雷)程序中,绝大多数的体数据以 1.2 节中提到的间接索引方式,通过面循环实现对体数据的修改,从而导致多个不同的面对同一个体数据进行修改。在 GPU 计算中,多个线程对应的面很可能同时对相同的体数据进行修改操作,导致错误的计算结果。这类计算需要首先解决资源竞争问题,才能在 GPU 上得到正确的计算结果。本文统计了需要解决资源竞争问题的计算占比,如表 1 所示,资源竞争问题的计算时间占总计算时间的 50% 以上,极大地影响了计算性能。图染色方法可以有效地解决资源竞争问题,需研究其性能及其优化。本文从 NNW-PHengLEI(风雷)程序中抽取了两种典型的算法讨论。

表 1 资源竞争问题计算时间占总时间的比例

Tab. 1 Proportion of race condition in overall performance

编号	计算描述	资源竞争操作	时间比例/%
C_1	变量插值	累加操作	15.90
C_2	梯度计算	累加操作	13.90
C_3	限制器计算	累加操作	17.91
C_4	限制器计算	局部最小值	3.25
C_5	限制器计算	局部最大值	3.25
C_6	残差更新	累加操作	5.22

算法 1 描述了由通量 *flux*(面数据)累加计算残差 *res*(体数据)的过程,属于非定常项计算中的通量累加更新残差计算部分(如图 1、表 1 中的 C_6 所示)。该算法通过遍历所有的面(0 ~ *numFaces* - 1),将面数据 *flux* 累加到 *owner*、*neighbor* 间接索引的体变量 *res*。结合图 2 可见,GPU 的多个线程在处理不同的面(f_1、f_2、f_3、f_4)时,有可能同时修改 $res[O]$,造成资源竞争。

算法 1 通量累加计算

Alg. 1 Computing of flux summation

```
1. for faceID = 0 to numFaces - 1 do
2.     ownVolID←owner[faceID]
3.     ngbVolID←neighbor[faceID]
4.     res[ownVolID]←res[ownVolID] + flux[faceID]
5.     res[ngbVolID]←res[ngbVolID] - flux[faceID]
6. end for
```

算法 2 描述了局部最大压力 *plmax*(体数据)的计算过程,属于对流通量计算中的限制器计算部分(如图 1、表 1 中 C_5 所示)。该算法通过遍历所有的面,借助 *owner*、*neighbor* 间接索引 *plmax*、*ps*,完成比较。由图 2 可知,$plmax[O]$ 有可能被 GPU 处理 f_1、f_2、f_3、f_4 面的多个线程同时修改,造成资源竞争。

算法 2 局部最大压力计算

Alg. 2 Computing of local maximum pressure

```
1. for faceID = 0 to numFaces - 1 do
2.     ownVolID←owner[faceID]
3.     ngbVolID←neighbor[faceID]
4.     plmax[ownVolID]←max(plmax[ownVolID],
ps[ngbVolID])
5.     plmax[ngbVolID]←max(plmax[ngbVolID],
ps[ownVolID])
6. end for
```

2 图染色算法及其应用

算法 1 和算法 2 在 GPU 计算中的资源竞争问题可以通过图染色方法解决。首先,将存在冲突的面进行染色分组,保证每一组中的任何两个面对应的计算不会修改相同的体数据;其次,将每一组计算分别放在 GPU 上,并通过全局内存读写的优化,降低访存延迟。

2.1 基于贪婪算法的面染色

面染色是一个 NP 完全问题[12]。在计算流体力学应用中,最优的染色结果只存在于简单的二维几何拓扑中[10]。对于复杂的三维几何拓扑,仅能够得到一般的染色结果[13]。本文采用贪婪算法对应用在空气动力学中的复杂三维几何拓扑进

行面染色分组。

首先,计算面的冲突关系。位于同一个体中的面相互冲突,计算面的冲突关系就是收集与该面同体的其他面。计算过程中除了利用 *owner*、*neighbor*,还要利用表征体 - 面拓扑的变量 *volFaces*、*volFacesPosi*。*volFaces* 存储了每个体包含的面数量以及面编号;*volFacesPosi* 存储了每个体的拓扑信息在 *volFaces* 中的位置。如图 2 所示,编号为 O 的体包含编号为 f_1、f_2、f_3、f_4 的 4 个面。O 对应的拓扑信息存储在 *volFaces* 中,起始位置为 *volFacesPosi*[O]。O 包含的面数量可以表示为 *volFaces*[*volFacesPosi*[O]] = 4,对应的面编号可以表示为 *volFaces*[*volFacesPosi*[O] + m] = f_m(m = 1,2,3,4)。因此,在包含 f_2 的编号为 *owner*[f_2] = O 的体中,存在与 f_2 冲突的面 f_1、f_3、f_4;类似地,编号 *neighbor*[f_2] = P 的体包含的面分别为 f_5、f_2、f_6、f_7,其中 f_5、f_6、f_7 与 f_2 冲突。最终,收集 f_2 的冲突关系,包括 f_1、f_3、f_4、f_5、f_6、f_7。汇集所有面的冲突关系,存储在变量 *nfcf*、*faceConflict*、*faceConflictPosi* 中,分别对应冲突面个数、冲突面 ID、冲突面的起始位置。f_2 的冲突信息表示为 *nfcf*[f_2] = 6,*faceConflict*[*faceConflictPosi*[f_2] + m](m = 0,1,2,3,4,5)分别对应了 f_1、f_3、f_4、f_5、f_6、f_7。具体描述见算法 3。

接着,利用冲突关系,通过基于贪婪算法的面染色方法将所有面进行染色,避免有冲突关系的面颜色相同,具体描述见算法 4。对于编号 *faceID* 的面,通过 *faceConflict*[*faceConflictPosi*[*faceID*] + m](m = 0, …, *nfcf*[*faceID*] - 1)遍历所有与 *faceID* 冲突的面,根据这些面已经染的颜色判断出面 *faceID* 可以染色的最小值。染色的结果存储在变量 *numFaceGroups* 和 *colorFaces* 中。*numFaceGroups* 表示颜色的种类;*colorFaces* 存储了每个面的颜色。

最后,将不同颜色的面分组。利用面染色结果 *numFaceGroups* 和 *colorFaces*,将相同颜色的面连续排列,存储在变量 *faceGroup* 中。同时记录每种颜色对应的面数量 *groupNum* 和每种颜色的面在变量 *faceGroup* 中的起始位置 *colorPosi*。

2.2　图染色分组计算的 GPU 算法

在面染色分组之后,每个组可以分别在 GPU 上计算。为了提高数据的局部性,减少数据读写的内存访问延迟,分别采用 GPU 共享内存、体和面编号重排、面数据重排的策略优化图染色方法。

算法 3　面的冲突关系计算

Alg. 3　Computing of getting conflicting faces

```
1.  sumFaceConflict←0
2.  for faceID = 0 to numFaces - 1 do
3.    ownVolID←owner[faceID]
4.    ngbVolID←neighbor[faceID]
5.    posiFaceConflict[faceID]←sumFaceConflict
6.    ownVolFaceID←posiVolFaces[ownVolID]
7.    sumFaceConflict←volFaces[ownVolFaceID] - 1
8.    ngbVolFaceID←posiVolFaces[ngbVolID]
9.    sumFaceConflict←volFaces[ngbVolFaceID] - 1
10.   nfcf[faceID]←sumFaceConflic
11. end for
12.  for faceID = 0 to numFaces - 1 do
13.       offsetFaceConflict←0
14.       faceConflictPosi←posiFaceConflict[faceID]
15.       ownVolID←owner[faceID]
16.       ownVolFaceID←posiVolFaces[ownVolID]
17.       numOwnVolFaces←volFaces[ownVolFaceID]
18.       for offset = 1 to numOwnVolFaces + 1 do
19.           ownFaceID ← volFaces [ ownVolFaceID +
offset]
20.         if ownFaceID ! = faceID then
21.           faceConflict [ faceConflictPosi +
offsetFaceConflict]←ownFaceID
22.           offsetFaceConflict←offsetFaceConflict + 1
23.         end if
24.       end for
25.       ngbVolID←neighbor[faceID]
26.       ngbVolFaceID←posiVolFaces[ngbVolID]
27.       numNgbVolFaces←volFaces[ngbVolFaceID]
28.         for offset = 1 to numNgbVolFaces + 1 do
29.           ngbFaceID←volFaces[ngbVolFaceID + offset]
30.             if ngbFaceID ! = faceID then
31.               faceConflict [ faceConflictPosi +
offsetFaceConflict]←ngbFaceID
32.               offsetFaceConflict←offsetFaceConflict + 1
33.             end if
34.         end for
35.  end for
```

算法 5 实现了通量累加(算法 1)的 GPU 计算。通过染色分组数量 *numFaceGroups* 确定循环次数,将每个分组依次放入 GPU 进行计算,避免了资源竞争。借助面染色分组信息 *groupNum*、

算法4 基于贪婪算法的面染色

Alg. 4 Face coloring based on greedy scheme

```
1. for faceID = 0 to numFaces - 1 do
2.     colorFaces[faceID] ← -1
3. end for
4. for faceID = 0 to numFaces - 1 do
5.     faceColorFlag←1
6.     color←0
7. while faceColorFlag do
8.       colorEqualFlag←0
9.       numConflictFaces←nfcf[faceID]
10.    for faceConflict = 0 to numConflictFaces - 1 do
11.       facePosi←faceConflictPosi[faceID] + faceConflict
12.       faceIDConflict←faceConflict[facePosi]
13.       if colorFaces[faceIDConflict] = color then
14.          colorEqualFlag←1
15.       end if
16.    end for
17.          if colorEqualFlag then
18.             color←color + 1
19.          else
20.             faceColorFlag←0
21.             colorFaces[faceID]←color
22.          end if
23. end while
24. end for
```

算法5 通量累加的图染色算法

Alg. 5 Face coloring for flux summation

```
1. for faceGroupID = 0 to numFaceGroups - 1 do
2.    Set numFacesInGroup←GroupNum[faceGroupID]
3.    Set colorStart←colorPosi[faceGroupID]
4.    < GPU kernel Begin >
faceInGroupID = threadIdx. x + blockIdx. x * blockDim. x
5.    if faceInGroupID < numFacesInGroup then
6.       Set colorID←colorStart + faceInGroupID
7.       Set faceID←faceGroup[colorID]
8.       Set ownVolID←owner[faceID]
9.       Set ngbVolID←neighbor[faceID]
10.      Set res[ownVolID]←res[ownVolID] + flux[faceID]
11.      Set res[ngbVolID]←res[ngbVolID] - flux[faceID]
12.   end if
13.    < GPU kernel End >
14. end for
```

colorPosi，可以得到每个分组对应的面数量 *numFacesInGroup* 和分组在 *faceGroup* 的起始位置 *colorStart*。接着，GPU 线程通过 *faceGroup* 得到面编号 *faceID*，再借助 *owner* 和 *neighbor* 得到相应的体编号 *ownVolID* 和 *ngbVolID*，将面上的通量 *flux*[*faceID*] 累加到相应的体数据 *res*[*ownVolID*] 和 *res*[*ngbVolID*] 中。

算法6实现了局部最大压力(算法2)的 GPU 计算。该算法借助染色分组信息，将分组依次放入 GPU 计算，避免了资源竞争。GPU 线程编号 *faceInGroupID* 通过 *faceGroup* 和 *colorStart*，实现了与面编号 *faceID* 的映射，并通过 *owner* 和 *neighbor*，得到相应的体编号 *ownVolID* 和 *ngbVolID*。最终，实现了体数据 *ps*[*ownVolID*] 与 *plmax*[*ngbVolID*] 的比较以及 *ps*[*ngbVolID*] 与 *plmax*[*ownVolID*] 的比较。

算法6 局部最大压力计算的图染色算法

Alg. 6 Face coloring for local maximum pressure

```
1. for faceGroupID = 0 to numFaceGroups - 1 do
2.    Set numFacesInGroup←groupNum[faceGroupID]
3.    Set colorStart←colorPosi[faceGroupID]
4.    < GPU kernel Begin >
      faceInGroupID = threadIdx. x + blockIdx. x * blockDim. x
5.    if faceInGroupID < numFacesInGroup then
6.       Set colorID←colorStart + faceInGroupID
7.       Set faceID←faceGroup[colorID]
8.       Set ownVolID←owner [faceID]
9.       Set ngbVolID←neighbor [faceID]
10.      Set plmax[ownVolID]←max(plmax[ownVolID],
ps[ngbVolID])
11.      Set plmax[ngbVolID]←max(plmax[ngbVolID],
ps[ownVolID])
12.   end if
13.    < GPU kernel End >
14. end for
```

在算法5和算法6中，GPU 线程根据面编号 *faceID*，通过 *owner* 和 *neighbor* 间接索引到编号分别为 *ownVolID* 和 *ngbVolID* 的体数据。*ownVolID* 和 *ngbVolID* 很难同时保证体数据读写的对齐。此外，染色分组造成了分组中面数据的非对齐访问。因此，加剧了 GPU 全局内存访问的延迟。

采用共享内存优化图染色法：针对间接索引带来的体数据无法对齐读取的问题，利用共享内存减小体数据读取延迟，相应的描述在算法7、算法8中。

算法7采用 GPU 共享内存对通量累加的图染色算法(算法5)进行优化，以减小数据读取延迟。该算法首先在 GPU 中定义共享内存变量 *fluxShare*。接着，GPU 线程先将面数据 *flux* 读取到 *fluxShare* 中，再将 *fluxShare* 累加到体变量 *res* 中。

算法 7　共享内存优化图染色算法(通量累加)

Alg. 7　Graph coloring method with optimization of shared memory (flux summation)

```
1. for faceGroupID = 0 to numFaceGroups - 1 do
2.  Set numFacesInGroup←groupNum[faceGroupID]
3.  Set colorStart←colorPosi[faceGroupID]
4.  < GPU kernel Begin >
5.  shared-memory fluxShare( )
faceInGroupID = threadIdx. x  +  blockIdx. x  *  blockDim. x
6.   if faceInGroupID < numFacesInGroup then
7.      Set colorID←colorStart + faceInGroupID
8.      Set faceID←faceGroup[colorID]
9.      Set ownVolID←owner[faceID]
10.     Set ngbVolID←neighbor[faceID]
11.     Set fluxShare( )←flux[faceID]
12.     Set res[ownVolID]←res[ownVolID] + fluxShare( )
13.     Set res[ngbVolID]←res[ngbVolID] - fluxShare( )
14.  end if
15.  < GPU kernel End >
16. end for
```

算法 8 利用 GPU 共享内存优化局部最值的图染色算法(算法 6),以减小数据读取延迟。该算法先将体变量 *ps* 读取到共享内存 *psShare* 中,再利用 *psShare* 和体数据 *plmax* 进行比较。

算法 8　共享内存优化图染色算法(局部最值)

Alg. 8　Graph coloring method with optimization of shared memory (local maximum pressure)

```
1. for faceGroupID = 0 to numFaceGroups - 1 do
2.  Set numFacesInGroup ←groupNum[faceGroupID]
3.  Set colorStart←colorPosi[faceGroupID]
4.  < GPU kernel Begin >
5.  shared-memory psShare( )
faceInGroupID = threadIdx. x + blockIdx. x * blockDim. x
6.   if faceInGroupID  <  numFacesInGroup then
7.      Set colorID←colorStart + faceInGroupID
8.      Set faceID←faceGroup[colorID]
9.      Set ownVolID←owner[faceID]
10.     Set ngbVolID←neighbor[faceID]
11.     Set psShare( )[0]←ps[ngbVolID]
12.     Set psShare( )[1]←ps[ownVolID]
13.     Set plmax[ownVolID]←max(plmax[ownVolID],
psShare( )[0])
14.     Set plmax[ngbVolID]←max(plmax[ngbVolID],
psShare( )[1])
15.  end if
16. < GPU kernel End >
17. end for
```

针对体数据的非对齐读写,可以通过重排体编号、面编号的策略[14]进行优化。重排后的编号使得面数量最大(即 *numFaceGroups* 最大)的分组实现体数据的对齐读写。通过提高数据的局部性,以减少 GPU 全局内存的访问延迟,相应算法描述在算法 9、算法 10 中。

算法 9　体、面编号重排优化图染色算法(通量累加)

Alg. 9　Graph coloring method with optimization of face and volume reordering (flux summation)

```
1. Get ownerRe and neighborRe by volume renumber
2. Get faceGroupRe by face reorder
3. for faceGroupID = 0 to numFaceGroups - 1 do
4.  < GPU kernel Begin >
faceInGroupID = threadIDx. x + blockIDx. x * blockDim. x;
5.  if faceInGroupID  <  numFacesInGroup then
6.     Set colorID←colorStart + faceInGroupID
7.     Set faceID←faceGroupRe[colorID]
8.     Set ownVolID←ownerRe[faceID]
9.     Set ngbVolID←neighborRe[faceID]
10.    Set res[ownVolID]←res[ownVolID] + flux[faceID]
11.    Set res[ngbVolID]←res[ngbVolID] - flux[faceID]
12.  end if
13.  < GPU kernel End >
14. end for
```

算法 10　体、面编号重排优化图染色算法(局部最值)

Alg. 10　Graph coloring method with optimization of face and volume reordering (local maximum presure)

```
1.  Get ownerRe and neighborRe by volume renumber
2.  Get faceGroupRe by face reorder
3.  for faceGroupID = 0 to numFaceGroups - 1 do
4.     Set numFacesInGroup←groupNum[faceGroupID]
5.     Set colorStart←colorPosi[faceGroupID]
6.     < GPU kernel Begin >
faceInGroupID = threadIdx. x + blockIdx. x * blockDim. x
7.   if faceInGroupID  <  numFacesInGroup then
8.      Set colorID←colorStart + faceInGroupID
9.      Set faceID←faceGroupRe[colorID]
10.     Set ownVolID←ownerRe[faceID]
11.     Set ngbVolID←neighborRe[faceID]
12.     Set plmax[ownVolID]←max(plmax[ownVolID],
ps[ngbVolID])
13.     Set plmax[ngbVolID]←max(plmax[ngbVolID],
ps[ownVolID])
14.  end if
15.     < GPU kernel End >
16. end for
```

算法9对通量累加的图染色法GPU计算(算法5)进行了优化。与算法5相比,对 *owner*、*neighbor*、*faceGroup* 进行了重排。重排后的 *ownerRe*、*neighborRe*、*faceGroupRe* 可以保证体编号 *ownVolID* 和 *ngbVolID* 在计算量最大分组中的连续性,使得体数据 *res* 的读写对齐。

算法10对局部最值的图染色法GPU计算(算法6)进行了优化。通过体、面编号排,实现了面数量最多分组中体数据 *plmax*、*ps* 读写的对齐。

最后,针对面数据的非对齐读取,重排面数据,使得面数据按照染色分组后的面编号重新排列,实现了面数据的对齐读取。相应的描述见算法11。

算法11对通量累加的染色法GPU计算(算法5)进行了优化。面数据 *flux* 是按照面标号递增的顺序排列的,但是经过染色分组后,分组中的面编号排序(*faceGroup*)并不是按顺序的,造成了面数据的非对齐读写。按照 *faceGroup* 中的面编号顺序重排 *flux*,得到面数据 *fluxRe*,将 *fluxRe* 累加到相应的体数据 *res* 中。

算法11 面数据重排优化图染色算法(通量累加)

Alg. 11 Graph cloring method with optimization of face reordering (flux summation)

```
1. ———————— Begin face variables reorder ————————
2. for faceGroupID = 0 to numFaceGroups - 1 do
3.    Set numFacesInGroup←groupNum[faceGroupID]
4.    Set colorStart←colorPosi[faceGroupID]
5.    for faceInGroupID = 0 to numFacesInGroup - 1 do
6.       Set colorID←colorStart + faceInGroupID
7.       Set faceID←faceGroup[colorID]
8.       Set fluxRe[colorID]←flux[faceID]
9.    end for
10. end for
11. ———————— End face variables reorder ————————
12. for faceGroupID = 0 to numFaceGroups - 1 do
13.    Set numFacesInGroup←groupNum[faceGroupID]
14.    Set colorStart←colorPosi[faceGroupID]
15.    < GPU kernel Begin >
faceInGroupID = threadIdx.x + blockIdx.x * blockDim.x
16.    if faceInGroupID < numFacesInGroup then
17.       Set colorID←colorStart + faceInGroupID
18.       Set faceID←faceGroup[colorID]
19.       Set ownVolID←owner[faceID]
20.       Set ngbVolID←neighbor[faceID]
21.       Set res[ownVolID]←res[ownVolID] + flux[faceID]
22.       Set res[ngbVolID]←res[ngbVolID] - flux[faceID]
23.    end if
24.    < GPU kernel End >
25. end for
```

3 实验结果与分析

3.1 算例和计算环境

为了分析图染色及其优化算法的性能,采用ONERA M6外流场CFD计算的三维非结构网格展开性能测试。如图3所示,以正四面体和棱柱体网格填充三维计算域,棱柱体网格存在于M6机翼附近。并采用如表2所示的5种不同网格规模测试算法的性能。

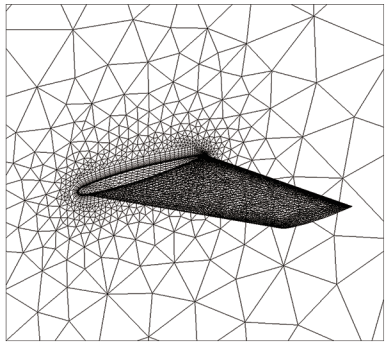


图3 ONERA M6网格

Fig. 3 ONERA M6 mesh

表2 5种网格规模

Tab. 2 5 different meshes

网格种类	体数量	面数量
网格1	0.46×10^6	1.04×10^6
网格2	0.89×10^6	1.91×10^6
网格3	1.96×10^6	4.04×10^6
网格4	3.92×10^6	8.02×10^6
网格5	9.76×10^6	20.28×10^6

算法5~11通过CUDA C实现。原CFD程序可以进行单、双两种精度的计算,因此本文分别针对双精度操作数和单精度操作数进行测试。每个kernel都重复运行1 000次并记录运行时间。

测试平台包括CUDA 10.0驱动的Nvidia Tesla V100 GPU以及CUDA 8.0驱动的Nvidia Tesla K80 GPU。

3.2 面染色结果

基于贪婪算法的面染色结果如表3所示,可见所有网格都被分为了8组。混合网格中含有棱柱体,棱柱体网格具有5个面,因此,面染色结果至少存在5个分组。对于复杂的三维几何拓扑,比理想化的分组数量多2~3个分组是可以接受的[15]。

表 3　面染色分组

Tab. 3　Results of face coloring (V100, double precision)

单位:个

分组	网格 1	网格 2	网格 3	网格 4	网格 5
1	224 836	426 256	921 933	1 858 291	4 628 854
2	213 031	410 166	896 907	1 817 425	4 370 933
3	206 237	397 696	871 117	1 767 691	4 223 509
4	197 699	383 289	841 903	1 709 925	4 064 937
5	163 051	239 361	433 943	747 020	2 377 710
6	40 517	50 293	76 778	117 274	587 202
7	1 482	1 834	2 091	2 399	26 444
8	11	21	27	7	458

3.3　算例运行时间

算法 5 ~ 11 对应的 GPU kernel 在 V100 上的运行时间如表 4 和表 5 所示。

表 4　运行时间(V100,双精度)

Tab. 4　Executing time (V100, double precision)

单位:s

算法编号	网格 1	网格 2	网格 3	网格 4	网格 5
算法 5	0.222	0.412	1.071	2.440	5.991
算法 6	0.198	0.378	1.067	2.564	6.310
算法 7	0.221	0.409	1.060	2.379	5.841
算法 8	0.196	0.372	1.032	2.458	6.090
算法 9	0.229	0.438	1.233	3.039	7.285
算法 10	0.203	0.406	1.424	3.778	8.750
算法 11	0.172	0.303	0.786	1.856	4.691

表 5　运行时间(V100,单精度)

Tab. 5　Executing time (V100, single precision)

单位:s

算法编号	网格 1	网格 2	网格 3	网格 4	网格 5
算法 5	0.159	0.279	0.673	1.519	3.927
算法 6	0.151	0.263	0.727	1.736	4.452
算法 7	0.158	0.277	0.666	1.481	3.829
算法 8	0.150	0.263	0.726	1.733	4.444
算法 9	0.167	0.293	0.724	1.856	4.906
算法 10	0.157	0.292	0.849	2.485	6.375
算法 11	0.133	0.232	0.550	1.251	3.350

算法 5 ~ 11 对应的 GPU kernel 在 K80 上的运行时间如表 6 和表 7 所示。

3.4　图染色方法在通量累加计算的性能比较

通过面染色分组,图染色方法可以消除 GPU 计算中的资源竞争。在面染色分组后,为了提高数据在 GPU 内存中的局部性,降低内存读写的延迟。针对累加操作,本研究采用共享内存、体编号和面编号重排、面数据重排对图染色方法进行了优化。下面进一步探讨这些优化方法的效果。

表 6　运行时间(K80,双精度)

Tab. 6　Executing time (K80, double precision)

单位:s

算法编号	网格 1	网格 2	网格 3	网格 4	网格 5
算法 5	1.108	2.182	5.019	10.483	25.935
算法 6	1.155	2.359	5.642	12.092	29.835
算法 7	1.103	2.165	4.968	10.221	25.287
算法 8	1.118	2.307	5.552	11.930	29.427
算法 9	1.151	2.375	5.695	12.422	29.528
算法 10	1.242	2.731	6.842	15.309	35.872
算法 11	0.959	1.906	4.415	9.247	22.860

表 7　运行时间(K80,单精度)

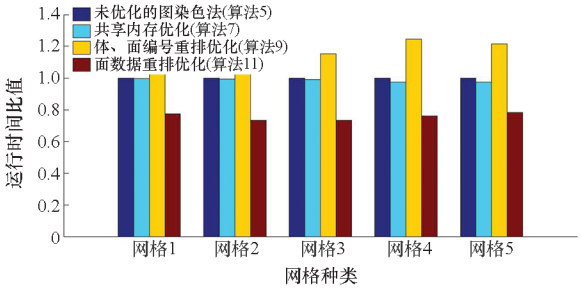
Tab. 7　Executing time (K80, single precision)

单位:s

算法编号	网格 1	网格 2	网格 3	网格 4	网格 5
算法 5	0.861	1.697	4.017	8.591	21.160
算法 6	0.851	1.801	4.508	9.957	24.425
算法 7	0.857	1.684	3.977	8.376	20.631
算法 8	0.890	1.848	4.584	10.074	24.763
算法 9	0.882	1.862	4.665	10.451	24.744
算法 10	0.900	2.116	5.647	13.162	30.555
算法 11	0.793	1.545	3.657	7.825	19.276

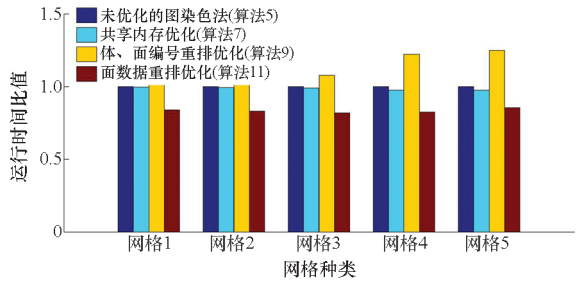
为了比较各种图染色方法在通量累加操作中的性能,以未优化的算法 5 运行时间为分母,得到算法 5(未优化)、算法 7(共享内存)、算法 9(体、面编号重排)、算法 11(面数据重排)的运行时间比值。

图 4 所示为 V100 的比较结果:共享内存的优化效果不明显,性能提高在 5% 以内;体编号和面编号重排使得执行时间增大;只有面数据 *flux* 重排的优化方法(算法 11)有效地减少了运行时间。双精度和单精度的计算结果近似,相比未加优化的图染色方法,面数据重排可以带来 20% 左右的性能提高。



（a）双精度

（a）Double precision



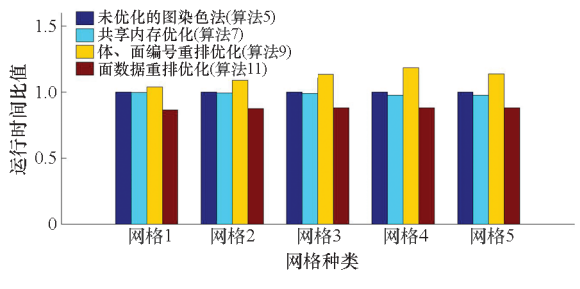
（b）单精度

（b）Single precision

图 4　图染色方法在通量累加操作中的性能(V100)

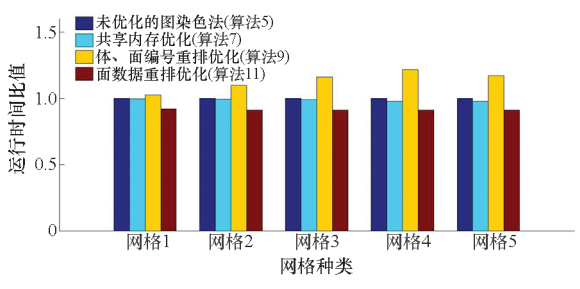
Fig. 4　Performance of graph coloring on flux summation operation (V100)

图 5 所示为 K80 的比较结果，和 V100 类似，只有面数据重排起到了优化效果：双精度计算的性能提升在 15% 左右，单精度计算的性能提升在 10% 左右。



（a）双精度

（a）Double precision



（b）单精度

（b）Single precision

图 5　图染色方法在通量累加操作中的性能(K80)

Fig. 5　Performance of graph coloring on flux summation operation (K80)

针对体编号、面编号重排造成计算性能下降的结果作了进一步分析。以网格 4(392 万个体、807 万个面)对应的 8 个染色组为例，分别在未优化(算法 5)和体、面编号重排(算法 9)2 种算法下计时，结果如表 8 所示。

可见，经过体编号、面编号重排后，第 1 组的运行时间由 0.640 s 降低到 0.241 s，这与第一组的面数据完全实现对齐相吻合。但是，第 2 ~ 7 组的运行时间全部增加，致使体、面编号重排后的总运行时间超过了未优化算法的运行时间。由此可见，体、面编号重排使得第 1 组数据的局部性得到提高的同时，恶化了其他几组的数据局部性。

表 8　网格 4 分组运行时间(V100，单精度)

Tab. 8　Executing time in groups of mesh 4 (V100, single precision)

单位：s

分组	未优化(算法 5)	体、面编号重排(算法 9)
1	0.640	0.241
2	0.564	0.998
3	0.504	0.948
4	0.492	0.929
5	0.288	0.543
6	0.097	0.126
7	0.008	0.009
8	0.006	0.006

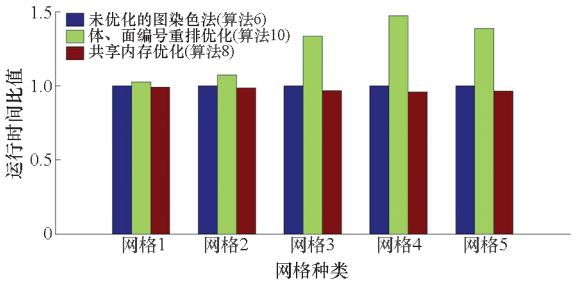
3.5　图染色方法在局部最大值计算的性能比较

NNW-PHengLEI(风雷)程序中存在一些局部最值操作，同样面临资源竞争问题。针对局部最大值计算，本研究采用共享内存、体编号和面编号重排这两种方法对图染色方法进行优化。

为了比较各种图染色方法在局部压力最大值计算中的性能，以未优化的算法 6 的运行时间为分母，得到算法 6(未优化)、算法 8(共享内存)、算法 10(体、面编号重排)的运行时间比值。

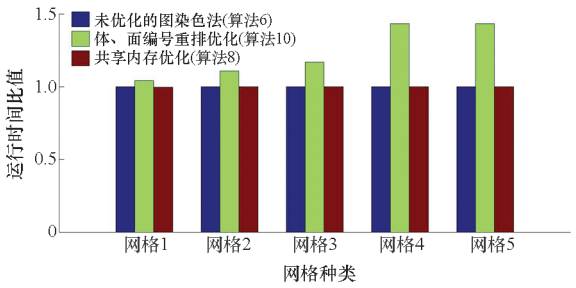
图 6 所示为 V100 的比较结果，共享内存优化方法的效果不明显；体编号、面编号重排使得运行时间增加；未优化的图染色法(算法 6)性能最佳。图 7 显示的 K80 比较结果与 V100 结果一致。

造成这一结果的原因主要在于局部最值运算仅操作体数据。面循环使得体数据访问的局部性降低。体、面编号重排或者共享内存都未能提高体数据的局部性。



(a) 双精度

(a) Double precision

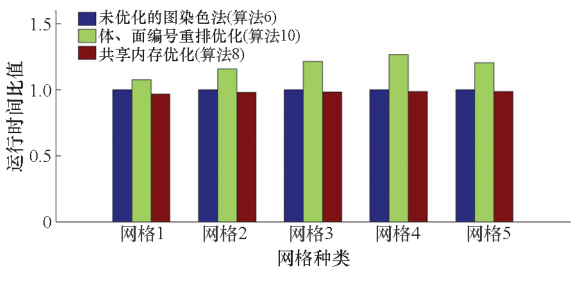


(b) 单精度

(b) Single precision

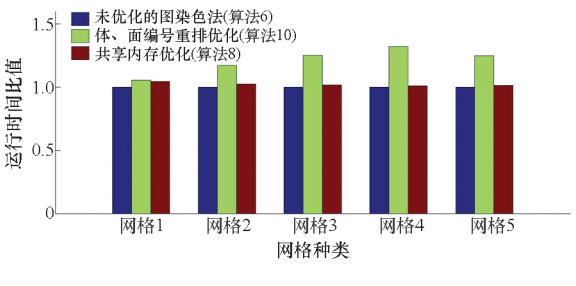
图 6　图染色方法在局部最大值操作中的性能(V100)

Fig. 6　Performance of graph coloring on local max operation (V100)



(a) 双精度

(a) Double precision



(b) 单精度

(b) Single precision

图 7　图染色方法在局部最大值操作中的性能(K80)

Fig. 7　Performance of graph coloring on local max operation (K80)

3.6　风雷程序 GPU 计算整体性能

本研究已经完成了 NNW-PHengLEI(风雷)程序非结构网格程序的异构开发和优化。图 8 展示了风雷程序 GPU 版本在 1 块 GPU(Nvidia Tesla V100)和 CPU 版本在 28 个 CPU 核(Intel Gold Xeon 6132)并行计算的加速比(双精度)。可见,在不同网格规模下,平均加速比达到 19。

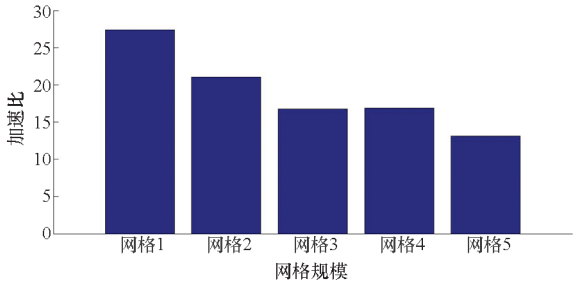


图 8　风雷程序 1 块 GPU 计算和 28 CPU 核计算的加速比

Fig. 8　Speedup of 1 GPU compared with 28 CPU cores

4　结论

本文采用图染色方法解决非结构网格有限体积计算流体力学在 GPU 计算中的资源竞争问题,并采用了共享内存、体编号和面编号重排、面数据重排的优化策略对图染色方法进行优化。通过图染色方法及其优化算法在 V100、K80 两种 GPU 上,针对应用在空气动力学中不同规模的复杂三维几何拓扑网格,测试通量累加计算和局部最大压力计算,得到如下结论:

1)对于通量累加计算,面数据重排可以将图染色方法在 GPU 上的计算性能提高 10%~20%;共享内存的优化作用小于 5%;体编号和面编号重排在提高 1 个分组数据局部性的同时恶化了其他分组的数据局部性,造成计算性能下降。

2)对于局部最大压力计算,共享内存和体、面编号重排都无法提高体数据的局部性。

今后,将针对不同的硬件架构,例如 AMD GPU、ARM、RISC-V 等展开图染色方法进行研究。

参考文献(References)

[1] ASHTON N, SKAPERDAS V. Verification and validation of OpenFOAM for high-lift aircraft flows[J]. Journal of Aircraft, 2019, 56(4): 1641 - 1654.

[2] STONE C P, WALDEN A, ZUBAIR M, et al. Accelerating unstructured-grid CFD algorithms on NVIDIA and AMD GPUs[C]// Proceedings of IEEE/ACM 11th Workshop on Irregular Applications: Architectures and Algorithms, 2021: 19 - 26.

[3] JONES T D, DAVIES D R, CAMPBELL I H, et al. The concurrent emergence and causes of double volcanic hotspot tracks on the Pacific plate[J]. Nature, 2017, 545(7655): 472 - 476.

[4] 赵钟, 张来平, 何磊, 等. 适用于任意网格的大规模并行 CFD 计算框架 PHengLEI[J]. 计算机学报, 2019,

42(11): 2368 – 2383.
ZHAO Z, ZHANG L P, HE L, et al. PHengLEI: a large scale parallel CFD framework for arbitrary grids[J]. Chinese Journal of Computers, 2019, 42(11): 2368 – 2383. (in Chinese)

[5] KOMATITSCH D, MICHÉA D, ERLEBACHER G. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA[J]. Journal of Parallel and Distributed Computing, 2009, 69(5): 451 – 460.

[6] FUHRY M, GIULIANI A, KRIVODONOVA L. Discontinuous Galerkin methods on graphics processing units for nonlinear hyperbolic conservation laws[J]. International Journal for Numerical Methods in Fluids, 2014, 76(12): 982 – 1003.

[7] GUO X H, LANGE M, GORMAN G, et al. Developing a scalable hybrid MPI/OpenMP unstructured finite element model[J]. Computers & Fluids, 2015, 110: 227 – 234.

[8] ZUBAIR M, NIELSEN E, LUITJENS J, et al. An optimized multicolor point-implicit solver for unstructured grid applications on graphics processing units[C]//Proceedings of 6th Workshop on Irregular Applications: Architecture and Algorithms, 2016: 18 – 25.

[9] ECONOMON T D, PALACIOS F, COPELAND S R, et al. SU2: an open-source suite for multiphysics simulation and design[J]. AIAA Journal, 2016, 54(3): 828 – 846.

[10] GIULIANI A, KRIVODONOVA L. Face coloring in unstructured CFD codes[J]. Parallel Computing, 2017, 63: 17 – 37.

[11] SULYOK A A, BALOGH G D, REGULY I Z, et al. Locality optimized unstructured mesh algorithms on GPUs[J]. Journal of Parallel and Distributed Computing, 2019, 134: 50 – 64.

[12] ZUCKERMAN D. Linear degree extractors and the inapproximability of max clique and chromatic number[C]// Proceedings of the thirty-eighth annual ACM symposium on Theory of computing-STOC '06, 2006: 681 – 690.

[13] ROKOS G, GORMAN G, KELLY P H J. A fast and scalable graph coloring algorithm for multi-core and many-core architectures[C]//Proceeding of Euro-Par 2015: Parallel Processing, 2015: 414 – 425.

[14] ZHANG X, SUN X, GUO X, et al. Re-evaluation of atomic operations and graph coloring for unstructured finite volume GPU simulations [C]//Proceedings of IEEE 32nd International Symposium on Computer Architecture and High Performance Computing, 2020: 297 – 304.

[15] XIA Y D, LUO L X, LUO H, et al. OpenACC-based GPU acceleration of a 3-D unstructured discontinuous Galerkin method [C]//Proceedings of 52nd Aerospace Sciences Meeting, 2014.