

GPU 平台上的叶轮机械 CFD 加速计算

鞠鹏飞, 宁方飞

(北京航空航天大学 能源与动力工程学院
航空发动机气动热力国家级重点实验室, 北京 100191)

摘 要: 通过数据并行的方式对一个成熟的叶轮机多块网格气动计算程序(MAP)进行了并行化处理, 利用计算统一设备架构(CUDA)技术实现了在图形处理单元(GPU)上的并行计算. 保留了原程序中的 2 阶空间迎风格式和隐式时间离散格式, 并采用了隐式迭代对线性系统进行求解. 经过 2 个叶轮机械算例的测试, 与在传统的中央处理器(CPU)上运行的原程序相比, 在计算结果完全一致的前提下, 单 GPU 的计算速度最高可达单 CPU 计算速度的 8.89 倍, 与四核并行的 CPU 计算相比可以得到 2.39 倍的加速.

关 键 词: GPU; CUDA; 并行计算; 隐式格式; 叶轮机械

中图分类号: V211.3

文献标志码: A

Accelerated CFD computing of turbomachinery on GPU platform

JU Peng-fei, NING Fang-fei

(National Key Laboratory of Science and Technology on Aero-Engine Aero-thermodynamics,
School of Energy and Power Engineering,
Beijing University of Aeronautics and Astronautics, Beijing 100191, China)

Abstract: The well-developed turbomachinery computational fluid dynamics code multi-block aerodynamic prediction (MAP) was solved by data-parallelized computing and implemented on a graphic processing unit (GPU) platform with the help of compute unified device architecture (CUDA) technology. The second-order upwind spatial scheme and the implicit temporal scheme in the original program were retained, while the linear system was solved by implicit iterations. During the test of two turbomachinery examples on a single GPU, a speed-up appears to be 8.89 times comparing with one central processing unit (CPU) process and 2.39 times comparing with four CPU processes, with no extra deviation induced into the result.

Key words: GPU; CUDA; parallel computing; implicit scheme; turbomachinery

计算流体力学(CFD)在现代航空航天科学技术中正发挥着越来越重要的作用,但是它对计算机的性能有着很强烈的需求,是典型的计算密集型学科. 由于单一的 CPU 性能有限,大规模的 CFD 计算都需要具有多个 CPU 的并行计算机来完成,并行计算的性能与成本都是与 CPU 数目成正比的,高性能的计算机往往体积巨大而且价格高昂,无法像普通个人计算机(PC)一样普及到

每一个 CFD 用户. 目前十分普及的 3-D 电子游戏同样需要巨大的计算量才能完成每一帧画面的渲染,但是由于现代 PC 引入了 GPU 用于图像处理,已经可以处理如此繁重的任务. 事实上, GPU 是一种集成了数百个算术运算核心的大规模并行计算芯片,它的峰值速度可达 CPU 的近 10 倍. 利用近几年兴起的图形处理器通用计算技术(GPGPU),已经可以使用 GPU 进行图形领域以外的

计算.目前主要的GPU厂商的产品都提供此类支持,如NVIDIA公司的CUDA,以及跨平台的开放计算语言(OpenCL)等,都是十分有效的GPGPU解决方案,并且已经在生物医药、天文、金融等很多领域内发挥了作用.由于PC主板可以安插多个GPU,一台PC最多可以拥有几十颗CPU的计算速度,成为一台个人超级计算机.当然,GPU本质上是并行的,所以适用于CPU的串/并行代码都不能直接应用于GPU,需要对代码进行修改,这在一定程度上限制了GPU的应用.

GPGPU发展的初期,人们就已经开始考虑将GPU强大的计算能力应用在CFD领域.早在2003年,Harris等^[1-2]就利用Cg语言(一种类似C语言的图形编程语言)及图形接口进行了二维及三维的Navier-Stokes层流数值模拟,在NVIDIA Geforce FX GPU上得到了相当于CPU 6倍的速度.2006年Hagen等^[3]利用Cg语言在NVIDIA 6800和7800 GPU上进行了二维和三维Euler方程的模拟,二维模拟得到20倍的加速,三维模拟得到了14倍的加速.而此后,BrookGPU和CUDA C这两种高级语言的出现,使得GPU编程计算的难度大大降低,在CFD方面的应用也越来越多.Brandvik和Pullan^[4-5]在2007年利用BrookGPU和CUDA进行了二维和三维Euler方程的模拟,在NVIDIA G80 GPU上实现了相当于Intel Core 2, 2.33GHz CPU单核16倍的运行速度;而Kestener等^[6]对二维Euler方程在NVIDIA TESLA S1070系统上用CUDA并行化获得了30~70倍的加速.在实际的工程计算中,也进行过高超声速飞行器(Elsen等^[7])、大气污染(Thibault等^[8-10]、Jacobsen等^[11])、现有程序移植(Overflow程序,Jespersen^[12])等工作,均能取得3~20倍的加速.在叶轮机械的气动数值模拟方面,Brandvik和Pullan^[13]在2009年对Denton的CFD进行了GPU移植并针对涡轮进行了数值模拟,在NVIDIA GT200 GPU上的运行速度达到了Intel Xeon 2.33GHz CPU上四核并行速度的20倍,这种加速效果是惊人的.近年来国内学者也在这方面进行了一些研究.中国科学院的王健等^[14]对方腔流的三维直接数值模拟(DNS)达到53倍的加速;中国科学院的董延星等^[15]进行的二维Euler方程模拟加速比为33.2,而N-S方程加速比为3.南京航空航天大学的张兵和韩景龙^[16-17]在结构和非结构网格上对三维机翼和YF-17机型进行了数值模拟,并采用了双

精度和隐式格式,达到了16~28倍的加速.上海交通大学苗树明^[18]的三维机翼绕流雷诺平均Navier-Stokes方程(RANS)计算的加速比为37.

本文基于课题组已有的三维结构网格RANS模拟程序MAP^[19],利用CUDA对其进行了移植使之适用于GPU,并保留了原程序的空间2阶迎风格式和时间推进的lower-upper symmetric successive over relaxation(LU-SSOR)隐式格式,从而保证了结果的可靠性.通过一个压气机转子和一个2级风扇算例的模拟,展示了该GPU版本的加速性能.

1 GPU架构与编程模型

由于目前计算核心的计算速度已经远超内存的响应速度百倍以上,内存的延迟已经成为拖累计算速度的最大障碍,也是架构优化最需要解决的问题.作为通用处理器,CPU需要承担的任务多种多样,所以CPU的性能优化主要针对一般的串行指令;而GPU面向单一的图形计算任务,能够对可并行的数据密集型运算进行专门的优化.图1表示了这两种优化的不同结果^[20];CPU选择了复杂的核心结构及大量的缓存(cache)和控制(control)元件,以预读取和高的缓存命中率来减少内存(DRAM)的延迟;而GPU则采用大量简单核心组成单指令多数据(SIMD)向量机架构,以长向量的合并访问以及多组线程的交替运行来减少和隐藏延迟;在同样的电路规模限制下,GPU配置了更多的算术计算核心(ALU),以此达到更高的峰值速度.图2给出了近年来高端CPU与高端GPU峰值速度逐年递增的水平^[20],图中1GFLOP/s表示每秒10亿次计算.



图1 GPU将更多晶体管直接用于数据运算

Fig. 1 GPU devotes more transistors to data processing

GPU架构取得高峰值性能的代价就是极大影响了架构的通用性,不恰当的算法会使计算速度严重下降甚至影响正确性.在编写CPU并行程序时,程序员最关注的是节点间避免通信和保

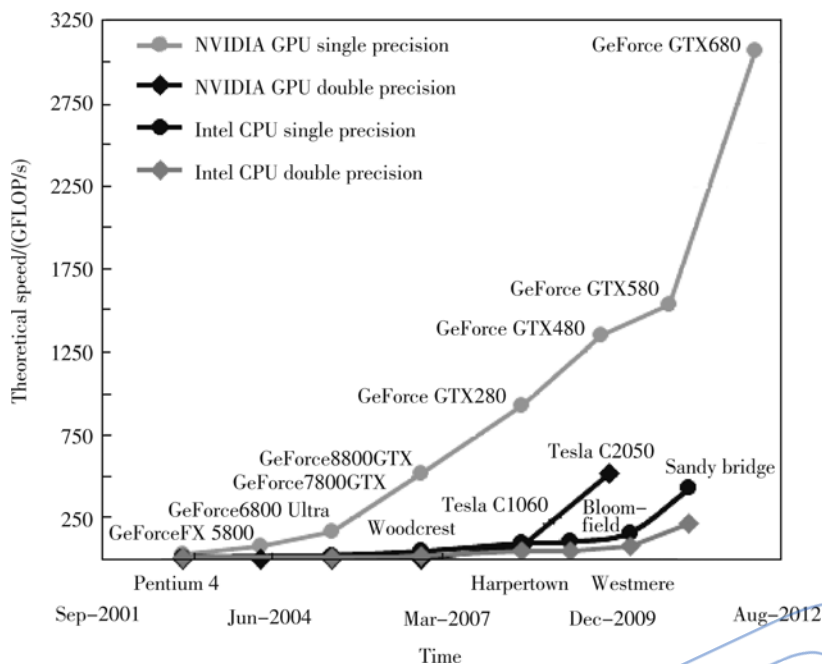


图 2 CPU 和 GPU 的理论浮点运算速度对比

Fig. 2 Theoretical floating-point operations per second for CPU and GPU

证负载平衡,对硬件架构则基本不必了解,主要的算法仍然是串行的.相比之下,为了使 GPU 达到理想的速度,程序的编写必须使得芯片上的各部分资源得到尽可能充分地利用,这需要程序员对 GPU 架构有一定程度的了解,并且始终带着并行的思想进行编程.尽管架构的相关性给 GPU 程序的移植带来一定的困难,但是针对硬件特点的合理优化往往能使程序获得更接近理论峰值的实际性能.

在 NVIDIA 的 CUDA 中使用的架构称为单指令多线程(SIMT)^[20].它在硬件上由多个结构相同的 SIMD 单元(CUDA 中称为 SM)组成,每个 SM 中都包含一定数目的计算核心(CUDA 中称为 CUDA 核心)、一组 SM 内的高速共享存储器以及公用的控制逻辑和存储控制器.除了上述芯片上的资源之外,GPU 还拥有一定数目的专用内存,通常与 GPU 一同焊接在电路板上,以扩展卡的形式出现(即市售的显卡),并通过 peripheral component interconnect express(PCI-Express)接口与主板交换数据.GPU 的内存带宽同样远高于 CPU,以支持高速的数据密集型计算.

由于 GPU 不具备独立运行操作系统的能力,整个程序还是由 CPU 发起,GPU 只是作为“设备”被 CPU 控制.GPU 所执行的代码(CUDA 中称为内核,kernel)作为 CPU 代码中的一个函数被调用,而内核代码描述的则是每个单一线程

的行为.除了内核之外,还有一些函数如 CPU 与 GPU 之间传递数据,以及状态查询、同步等,它们也同样被 CPU 代码所调用.在 GPU 处理内核期间,允许 CPU 继续执行下面的 CPU 代码,以实现更大程度的协同计算.

为了控制 SM 内的线程数,线程被组织成线程块(block)并在运行时整块分配给某个 SM,线程块数及每块中的线程数在内核调用时被显式指定.在运行时,SM 应该加载尽可能多的线程,使得一组线程进入延迟等待时很容易在队列中找到另一组准备就绪的线程继续操作,从而避免核心的闲置.与线程块相对应的还有块内的低开销同步机制,以及使用 SM 共享存储器的块内高速数据共享,这些都使得 GPU 的编程更加灵活,因而更容易接近峰值速度.

由于内核及其调用过程只定义线程及线程块的行为,同一程序可以兼容 SM 数目不同的 GPU,SM 数目更多的高端 GPU 因为同时处理更多的线程块所以消耗更短的时间,如图 3 所示^[20].内核中允许出现分支,但 SM 会依次处理所有分支而不是并行处理,这将严重影响效率,所以也应该尽量避免.

本文使用的 GPU 为 NVIDIA GTX 470,具有 14 个 SM,每个 SM 包含 32 个 CUDA 核心,与之搭配的 CPU 为 Intel Core2 Q9550,它有 4 个核心.表 1 列出这两种芯片及其内存的参数.

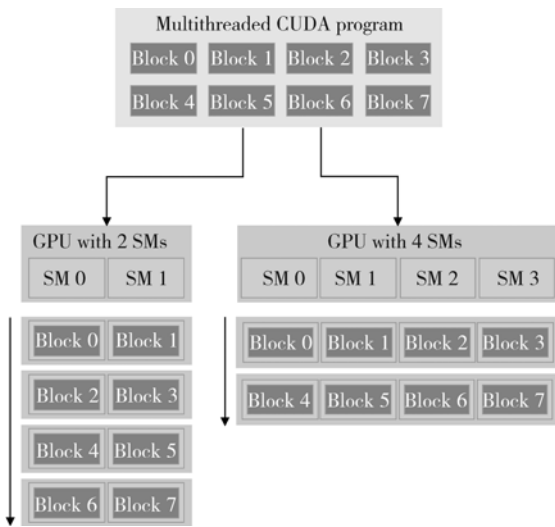


图 3 CUDA 程序的可伸缩性

Fig. 3 Automatic scalability of CUDA program

表 1 两种芯片及其内存的技术参数

Table 1 Technical specifications of two chips and memories

	GTX 470 GPU	Q9550 CPU
核心总数	448	4
核心频率/MHz	1215	2830
二级缓存/MB	0.625	12
内存容量/GB	1.25	4
内存带宽/(GB/s)	133.9	12.8

2 离散方式与并行化

在 MAP 程序中,控制方程为雷诺平均的 Navier-Stokes 方程,其空间积分形式可以写成

$$\frac{\partial}{\partial t} \int_V \mathbf{Q} dV + \oint_{\partial V} (\bar{\mathbf{F}} - \bar{\mathbf{F}}_v) \cdot \mathbf{n} dS = 0 \quad (1)$$

其中 $\mathbf{Q}$ 为守恒变量, $\bar{\mathbf{F}}$ 为对流量, $\bar{\mathbf{F}}_v$ 为扩散通量, V 和 S 分别为控制单元的体积和表面积. 式中暂不考虑体积力项.

2.1 空间离散方法

MAP 程序采用了单元中心的有限体积离散方法进行空间离散,将式(1)应用在控制体 ℓ 上,并假设 ℓ 中的物理量为常值,则有如下的积分形式:

$$\frac{\partial}{\partial t} (\mathbf{Q} \cdot \Delta V)_\ell + \sum_{k=1}^M [(\bar{\mathbf{F}} - \bar{\mathbf{F}}_v)_k \Delta S_k] = 0 \quad (2)$$

其中 ΔV 为控制单元的体积, M 为控制单元 ℓ 的表面数目,对本文所使用的三维结构网格, $M=6$; ΔS_k 为控制单元第 k 个表面的面积,向单元内为

正; $\bar{\mathbf{F}}$ 和 $\bar{\mathbf{F}}_v$ 分别为控制单元表面的对流和扩散通量. 为了取得更好的间断捕获能力, MAP 程序采用了低耗散通量分裂格式(LDFSS)对对流量 $\bar{\mathbf{F}}$ 进行空间离散,并使用 MUSCL(monotone up-stream-centered schemes for conservation laws)插值的方式达到 2 阶精度. 对于扩散通量 $\bar{\mathbf{F}}_v$ 的求解使用了相对简单的中心型格式.

2.2 时间离散方法

MAP 程序通过隐式算法对离散系统进行计算,所以使用的时间离散格式也是隐式的. 首先把式(2)改写为如下形式:

$$\frac{\partial}{\partial t} (\mathbf{Q} \cdot \Delta V)_\ell + \tilde{\mathbf{R}}(\mathbf{Q})_\ell = 0 \quad (3)$$

其中 $\tilde{\mathbf{R}}$ 为广义残差向量

$$\tilde{\mathbf{R}}(\mathbf{Q})_\ell = \sum_{k=1}^M [(\tilde{\mathbf{F}} - \tilde{\mathbf{F}}_v)_k \Delta S_k] \quad (4)$$

对式(3)采用 1 阶 Euler 后差格式进行时间离散,并省略控制体标号 ℓ , 有

$$\frac{\Delta V}{\Delta t} (\mathbf{Q}^{n+1} - \mathbf{Q}^n) + \tilde{\mathbf{R}}^{n+1} = 0 \quad (5)$$

其中上标 n 表示时间层. 将残差向量进行线性化处理,可得

$$\left[\frac{\Delta V}{\Delta t} \mathbf{I} + \frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{Q}} \right]^n \delta \mathbf{Q}^n = -\tilde{\mathbf{R}}^n \quad (6)$$

其中 $\delta \mathbf{Q}^n = \mathbf{Q}^{n+1} - \mathbf{Q}^n$ 是方程的未知量, $\frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{Q}}$ 是残差向量的线性化矩阵. 这是一个形如 $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$ 的线性方程组,对流动方程的求解转化为对线性方程组式(6)的求解.

2.3 线性系统的求解方法

为了提高数值格式的稳定性, MAP 程序使用了 LU-SSOR 对方程组式(6)进行求解,这种方法最早由 Jameson 和 Yoon^[21] 于 1987 年提出. 下面简要给出求解过程.

利用 Steger 和 Warming 的系数矩阵分裂方法,参照式(4)可以将控制单元表面处对流通量的线性化矩阵与解向量的乘积近似为

$$\left[\frac{\partial (\tilde{\mathbf{F}} \Delta S_k)}{\partial \mathbf{Q}} \delta \mathbf{Q} \right]_k \cong [\hat{\mathbf{A}}_1^+ \cdot \delta \mathbf{Q}_1 + \hat{\mathbf{A}}_r^- \cdot \delta \mathbf{Q}_r] \Delta S_k \quad (7)$$

其中下标 1 和 r 分别表示该面两侧控制体内的值, $\hat{\mathbf{A}}^\pm$ 是对流量分裂后的线性化矩阵,其中正矩阵为正定,负矩阵为负定,且都为主对角占优的.

在这里,有

$$\hat{\mathbf{A}}^{\pm} \Delta S = \frac{1}{2} (\hat{\mathbf{A}} \Delta S \pm r_a \mathbf{I}) \quad (8)$$

其中 r_a 大于或等于线化矩阵的谱半径. 对于扩散通量, 通过对该表面两侧的控制体内守恒变量求偏导数得到类似的形式

$$\left[\frac{\partial(\tilde{\mathbf{F}}_v \Delta S)}{\partial \mathbf{Q}} \delta \mathbf{Q} \right]_k \cong (\tilde{\mathbf{A}}_{vl}^* \cdot \delta \mathbf{Q}_l + \tilde{\mathbf{A}}_{vr}^* \cdot \delta \mathbf{Q}_r) \Delta S \quad (9)$$

以二维情形为例, 经过近似因式分解, 方程组式(6)可以展开为

$$(\mathbf{D} + \mathbf{L}) \mathbf{D}^{-1} (\mathbf{D} + \mathbf{U}) \Delta \mathbf{Q} = -\tilde{\mathbf{R}} \quad (10)$$

并由二步法来求解

$$(\mathbf{D} + \mathbf{L}) \Delta \mathbf{Q}^{(1)} = -\tilde{\mathbf{R}}^n$$

其中

$$(\mathbf{D} + \mathbf{U}) \Delta \mathbf{Q}^n = \mathbf{D} \Delta \mathbf{Q}^{(1)} \quad (11)$$

$$\begin{aligned} \mathbf{L} &= \bar{\mathbf{A}}_{i-1}^+ \Delta S_{i-\frac{1}{2}} + \bar{\mathbf{A}}_{j-1}^+ \Delta S_{j-\frac{1}{2}} \\ \mathbf{U} &= -\bar{\mathbf{A}}_{i+1}^- \Delta S_{i+\frac{1}{2}} - \bar{\mathbf{A}}_{j+1}^- \Delta S_{j+\frac{1}{2}} \\ \mathbf{D} &= \left[\frac{\Delta V}{\Delta t} + r_a \right] \mathbf{I} + (\tilde{\mathbf{A}}_v^+ - \tilde{\mathbf{A}}_v^-) \Delta S \\ \bar{\mathbf{A}}^{\pm} &= \hat{\mathbf{A}}^{\pm} - \tilde{\mathbf{A}}^{\pm} \end{aligned} \quad (12)$$

从式(12)可以看出: 二步法中的每一步都只用到单元某一侧的表面, 这说明每个单步中数据的依赖是单方向的, 而这种依赖正是隐式迭代法的基本特征. 文献[22]指出: 式(11)在结构化网格上的求解实际上是沿网格对角线方向的扫掠. 图 4^[22]表示了二维和三维结构网格计算域中求解的扫掠方向.

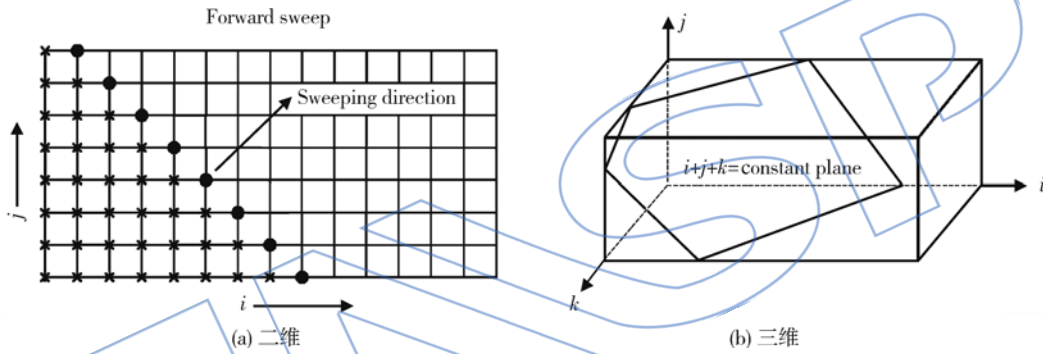


图 4 二维和三维情形下 LU-SSOR 方法的扫掠方向

Fig. 4 Sweeping direction of LU-SSOR scheme under circumstance of two-dimension and three-dimension

2.4 GPU 上的隐式并行方法

时间推进过程可以大致分为边界赋值、通量计算、方程求解 3 个过程. 为了较快地将 MAP 程序应用到 GPU 平台上, 在这里选择将耗时最长的通量计算和方程求解过程转移到 GPU 上进行, 而边界赋值过程由于代码量较大而耗时很短(一般不足总时间的 5%), 所以保留在 CPU 上进行.

串行程序在 GPU 上的并行化, 就是将循环体展开成大量独立线程的过程. CFD 程序是对离散系统的处理, 在串行程序中, 对离散单元(控制体或其表面)进行的每一次计算都以循环的形式针对不同的单元重复进行, 将单元的编号作为循环变量以遍历所有单元. 而 GPU 编程则是利用不同的线程对不同单元进行处理, 单元编号与线程编号是一一对应的. 只要循环中的前后步之间没有数据依赖, 这种循环的展开是很容易实现的. 在 MAP 程序中, 通量的计算不存在单元间的依赖, 它的并行化很容易地得到了实现.

相比之下方程求解的过程就难于做到并行化. 由 2.3 节的式(11)可以看出: 由于系数矩阵 $\mathbf{L}$ 和 $\mathbf{U}$ 的存在, 解向量的一个元素是需要利用相邻元素更新后的值来求出的, 这种依赖性使得数据并行无法实现. 要解决这个问题, 一般需要避开隐式推进的办法, 要么可以使用显式格式进行时间推进(文献[3, 6, 15]的做法), 要么可以使用显式迭代(如 Jacobi 迭代)方法对式(6)进行求解(文献[16-17]的做法), 因为显式方法天然满足数据并行的要求. 本文采取的方法则不相同, 通过推进面上局部并行的方法实现了隐式推进的并行化计算.

由图 4 可以看出: 求解的过程是逐个对角面(等 $i+j+k$ 面)推进的, 而由式(12)可知求解所需要的信息都位于相邻的对角面上, 所以在同一个对角面内各单元的求解是相互独立的, 可以将同一个对角面内的单元同时进行独立计算, 而按照扫掠的顺序依次进行各个对角面的计算, 就可以实现隐式的并行计算.

CUDA架构中没有提供整个GPU并行任务内的同步机制,所以只能将每个对角面作为一个独立的并行任务依次启动,并且在这些任务间插入CPU-GPU同步函数以确保顺序执行.在3.3节可以看到:这会在很大程度上影响执行效率.为了最大限度地减少并行任务数,在多块网格中对不同网格块的对角面并入同一任务,使并行任务数只取决于对角面数最多的网格块,所以下文提到的对角面数均指单块网格的对角面数最大值.

原程序中数据沿网格线排列,这使得沿对角面并行的任务中相邻线程的数据不连续,不能实现合并访问,而合并访问正是GPU实现高数据吞吐量的基础.为了维持数据的连续性,对有关的数组都沿对角面进行了重新排序.在一个初步的测试中,这种重排序使推进时间由1.0s缩短到0.5s,说明了重排序的重要性.

2.5 MPI-GPU 双重并行

MAP程序支持基于消息传递截面(MPI)的并行计算,并且所涉及到的负载分配及网格剖分对接都是自动完成的,无需用户针对并行计算定义任何文件,十分方便.为了维持原有的并行框架,GPU并行仅针对每一个MPI进程内的任务进行.每个进程都可以根据设备编号关联到指定的一个GPU,而同一个GPU可以被多个进程同时关联和使用.

为了使GPU计算平台的计算任务不受GPU存储器容量的限制,多进程共用GPU被组织成分时复用的方式.通过针对进程号的循环与同步,使该GPU同一时段只被一个进程所使用.在使用的这段时间内,该进程依次完成分配GPU存储器空间、向GPU传递全部数据、发送指令使GPU计算、由GPU回传结果、释放GPU存储器空间这一系列动作,其他时间则处于同步等待中.由于存储器空间被及时释放,GPU可以依次完成多个进程的计算任务.事实上,由于GPU存储器成本很高且不能自主替换,所以对于目前工程上动辄几百上千万单元的计算量来讲,这种分时复用是必须的.3.3节将要讲到这种额外的计算规模是以牺牲性能为代价的.

3 测试结果及分析

本文使用PGI FORTRAN编译器进行编译,它是目前唯一的完全基于FORTRAN编译器的GPGPU解决方案,同时支持CPU编程和CUDA架构的GPU编程.在为CPU编程时支持FOR-

TRAN 95的全部语法,而为GPU编程的语法也是基于FORTRAN语法的扩展.这里所使用的主要编译选项为pgfortran -fastsse -Mipa=fast,inline -Mmpi=msmpi -Mcuda=cc20,代码包含了CPU与GPU计算的功能,通过配置文件选择具体使用的硬件.所使用的CPU为Intel Core 2 Q9550四核四线程CPU,所使用的GPU为一个NVIDIA GTX 470显卡,包含448个CUDA核心,操作系统为Windows 7 64位版.选用的2个算例分别为一个压气机转子和一个2级风扇,主要考察计算速度的变化.所用湍流模型均为Spalart-Allmaras一方程模型,CPU和GPU均使用双精度计算.

尽管以下2个算例的规模都不必依赖分时复用来完成,但是为了体现分时复用对性能的影响,多进程的测试中均采用了分时复用的方式.

3.1 压气机转子算例

在一个100万网格的压气机转子定常计算算例上,测试了一块GPU的单进程、两进程和四进程时的加速效果,其中GPU在两进程和四进程时为分时复用方式.相应地,CPU计算则分别测试了单进程、两进程和四进程并行的速度,其中四进程时为该CPU的最大速度.该算例的网格分为7块,对角面最多的网格块为 $218 \times 27 \times 90$ 个单元,333个对角面.计算网格如图5所示.

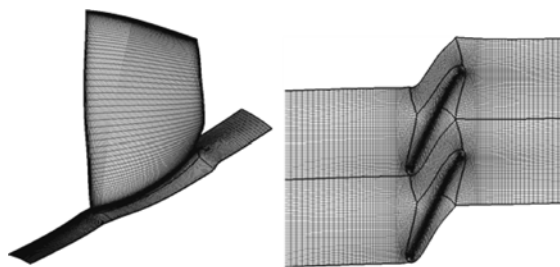


图5 压气机转子的计算网格

Fig. 5 Computational grid for compressor rotor

表2给出了这6种情况每一个时间步所花费的时间.从运行速度来看,单进程GPU计算的速度最高,单进程CPU的速度最低.如果对比单进程的运行速度,则加速比为7.72.如果对比GPU和CPU的最大速度,则加速比仅为2.39,这个值最能代表MAP程序向GPU扩展后在该测试平台上得到的性能提高.图6显示了单进程下残差随时间的收敛轨迹,可以看出GPU更早达到同等的收敛.

表 2 不同进程数下每步消耗的时间 (单位:s)

Table 2 Time consume per step for different process numbers (unit:s)

进程数	单步时间(CPU)	单步时间(GPU)
1	11.58	1.50
2	6.04	1.91
4	3.59	2.47

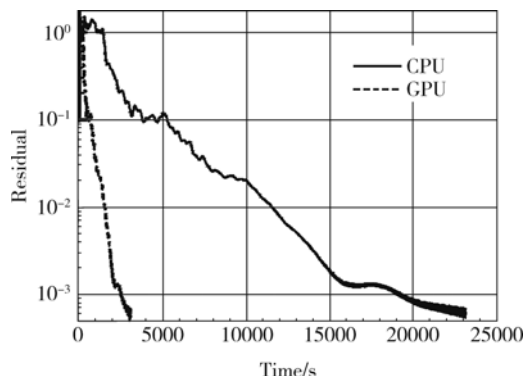


图 6 残差的收敛过程

Fig. 6 Convergence process of residual

对比单进程 GPU 与单进程 CPU 的执行过程,表 3 给出了几个主要的计算步骤所消耗的时间及加速比。可以看到:转移到 GPU 上的 2 个步骤都得到了加速,而由于数据传递带来的额外开销只占总时间的 6%,这种影响是比较小的。通量计算获得的加速比最大,而时间推进部分的加速比相对小一些,可见对角面内的局部并行方式对 GPU 的加速是不利的。

表 3 压气机转子单进程主要步骤运行时间

Table 3 Run time of main procedures in single process test of compressor rotor

	CPU 时间/s	GPU 时间/s	加速比
边界赋值	0.345	0.342	1.01
数据传递	0	0.097	
通量计算	6.179	0.350	17.65
时间推进	5.056	0.705	7.17
总时间	11.58	1.500	7.72

对比单进程 GPU 与多进程分时复用 GPU 更能看出局部并行带来的速度损失。从表 4 看出:当 100 万单元的计算量被均分到 2 个进程中分别计算时,总计算量没有增加,而总时间却变为了 1.28 倍,其中通量计算的时间基本没有增加,时

间推进的耗时则大幅增加。分成 2 个进程后,时间推进部分需要启动的内核数变为 2 倍,这说明大量内核启动导致额外消耗的时间在本算例中约为 0.3 s,这与整个时间步所花时间相比是不能被忽视的。此外,多进程分时复用还导致数据传输时间增大了 0.3 s,这使得总时间进一步增大,同样极大地影响了加速性能。

表 4 GPU 在不同进程数下主要步骤运行时间 (单位:s)

Table 4 Run time of main procedures for different process numbers on GPU (unit:s)

	单进程	二进程	四进程
边界赋值	0.342	0.107	0.107
数据传递	0.097	0.411	0.477
通量计算	0.350	0.367	0.408
时间推进	0.705	1.029	1.480
总时间	1.500	1.913	2.470

由于 GPU 与 CPU 使用了完全相同的数值格式,并且都能完全支持 IEEE754-2008 二进制浮点运算标准,所以程序移植并没有带来额外的误差。

3.2 2 级风扇算例

在另一个 2 级风扇定常计算的算例中,网格单元数同样为 100 万,该风扇包括进口导叶在内共 5 排叶片,平均每个通道只有 20 万网格,这是工程计算中常用的网格密度。这套网格与上个算例相比,最大的不同在于网格块的尺寸减小了,最大对角面数为 244,根据上个算例的经验,可以预期时间推进部分所用的时间将减少 0.1 s。

针对这个算例的单进程详细运行时间见表 5,可见同为 100 万单元的网格规模,本算例达到的加速比更高(8.89),而速度的加快是由于时间推进部分耗时减少了 0.14 s,与前面的预测大致相当。更高的速度进一步说明了对角面数对程序执行速度的影响。

表 5 2 级风扇单进程主要步骤运行时间

Table 5 Run time of main procedures in single process test of two-stage fans

	CPU 时间/s	GPU 时间/s	加速比
边界赋值	0.194	0.216	0.898
数据传递	0	0.103	
通量计算	5.863	0.360	16.29
时间推进	5.104	0.568	8.99
总时间	11.161	1.255	8.89

3.3 耗时因素分析

通过上述结果可以发现,计算过程中的3个低加速比的环节拉低了总的加速比,分别是时间推进、边界赋值及数据传递。

对于时间推进过程,前面已经提到,它的计算速度受到对角面个数的影响。图7给出了对更多算例测试结果的统计,可以明显发现每百万个单元的耗时随平均对角面大小的增大而减小。假设耗时随单元数和对角面都呈线性变化,根据这些样本线性回归可以得出:单由启动对角面并行任务带来的耗时约为每百个对角面0.15s,而真正用于计算的耗时约为每百万个单元0.21s(图中虚线所示),可见计算速度没有得到充分发挥。对于多余的耗时,首先怀疑是来自并行任务没有与SIMD宽度对齐所造成的计算核心浪费,然而根据CUDA的处理方式,在最极端的假设下每个对角面都有448个单元(GPU的核心数)的计算量被浪费,以此计算每百个对角面的浪费也只有约0.0094s,可见这不是耗时的主要来源。对于这种算法以外的额外开销,只能看作是硬件延迟,是任务结束后CPU与GPU间的同步造成的,需要通过硬件的进步来解决。新近上市的GK110架构GPU引入了内核发射内核的动态并行机制^[23],应该能够改善这种数据依赖情况下的并行效率。然而图7直接说明了大的平均对角面有利于减少计算能力的浪费,所以在一个进程上加载更多的计算量是一种更加经济的做法,这又会受到GPU内存规模的限制。

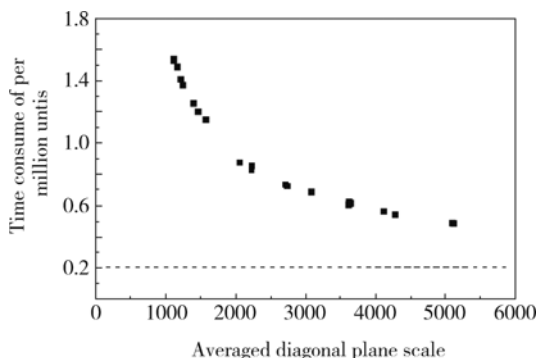


图7 每百万个单元耗时与平均对角面大小的关系

Fig. 7 Relationship between time consume per million units and averaged diagonal plane scale

数据的重排序是实现较高加速比的重要保证。尽管2.4节提到重排序使得时间推进耗时减少了约50%,但是考虑到上述延迟的存在,实际

上对计算速度的提升应该有数倍之多。通常在满足数据并行要求的任务中,线程展开的方向都取到与数据排列的方向相同,这种合并访问的要求是天然满足的。而在本文的时间推进过程中,线程方向受到几何特征的限制,所以只能靠改变数据排列方式来满足数据的连续性。通量计算及CPU部分的执行速度则不受重排序的影响。

由于边界赋值部分在MAP程序中的代码量大于通量计算与时间推进部分之和,而实际计算量却不及后者的十分之一,故优先对后者进行了移植。然而随着总时间的缩短,边界赋值耗时已经变得不可忽视。在前述2个算例中,边界赋值大约占用总时间的1/5,尚不足以构成性能的瓶颈,但是当采用多个高性能GPU与一个中等性能的CPU搭配时,这一比例可能发生显著变化,届时边界赋值过程将会严重拖累整体性能的提高。为了减轻对CPU性能的依赖,后期有必要对边界赋值工作也进行移植,或者进行大幅度的优化。

表3显示分时复用GPU会带来更长的数据传输时间,这是因为在分时复用GPU所用到的所有数组(主要是几何信息)都要在计算前传输一次,这些要远远多于需要刷新的流场解本身。执行传递的PCI-E 2.0 X16接口理论上具有8.0GB/s的带宽,但是本文所使用的平台实际只达到了2.0GB/s,有可能使用品质更好的硬件来缩短传输时间。已经上市的新一代GPU采用了具有16.0GB/s理论带宽的PCI-E 3.0 X16接口,传输时间有望进一步缩短。唯有避免GPU分时复用才能真正解决这个问题,而不分时复用所允许的计算规模同样受到GPU内存容量的限制。

4 结论

本文将一套成熟的叶轮机械CFD计算程序MAP移植到GPU上,通过对通量计算和线性系统求解过程的并行化处理,实现了GPU上的LU-SSOR格式隐式并行计算。在所使用的测试平台上,单进程的加速比最高可达8.89倍,而单GPU与四核并行CPU相比也有2.39倍的加速,并且完全没有造成计算精度的降低。经过2个叶轮机械算例的测试研究得到如下结论:

1) 针对本文所发展出的程序,应该尽量增大单进程的任务量,一方面有利于避免GPU被分时复用,另一方面可以在更大的计算量中分摊对角面启动延迟。故最好选用内存容量更大的GPU。

2) 在程序的进一步发展过程中,应该设法将

边界赋值的任务也交给 GPU 来完成,以减少这部分的运行时间以及在不分时复用的情况下压缩数据传递量。

3) GPU 的巨大速度优势是建立在充分数据并行的基础上的,所以应该着重发展基于数据并行的 CFD 数值方法,以彻底解决本文遇到的对角面启动延迟问题。

参考文献:

- [1] Harris M J. Fast fluid dynamics simulation on the GPU [C]// GPU Gems. Boston: Addison Wesley, 2004: 637-665.
- [2] Harris M J, Baxter W V, Scheuermann T, et al. Simulation of cloud dynamics on graphics hardware[C]// Proceedings of the ACM Siggraph/Eurographics Conference on Graphics Hardware. Aire-la-Ville, Switzerland: Eurographics Association, 2003: 92-101.
- [3] Hagen T R, Lie K A, Natvig J R. Solving the Euler equations on graphics processing units[R]. Berlin: International Conference on Computational Science, 2006.
- [4] Brandvik T, Pullan G. Acceleration of a 2D Euler solver using graphics hardware[R]. Cambridge University Engineering Department Technical Report, CUED/A-TURBO/TR. 132, 2007.
- [5] Brandvik T, Pullan G. Acceleration of a 3D Euler solver using commodity graphics hardware [R]. AIAA-2008-607, 2008.
- [6] Kestener P, Chateau F, Teyssier R. Accelerating Euler equations numerical solver on graphics processing units [J]. Lecture Notes in Computer Science, 2010, 6082: 281-288.
- [7] Elsen E, Legresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU[J]. Journal of Computational Physics, 2008, 227(24): 10148-10161.
- [8] Thibault J C, Senocak I. CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows[R]. AIAA-2009-758, 2009.
- [9] Thibault J C. Implementation of a Cartesian grid incompressible Navier-Stokes solver on multi-GPU desktop platforms using CUDA[D]. Boise: Boise State University, 2009.
- [10] Senocak I, Thibault J, Caylor M. Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputers[R]. Phoenix: 8th American Meteorological Society Symposium on the Urban Environment, 2009.
- [11] Jacobsen D A, Thibault J C, Senocak I. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters [R]. AIAA-2010-522, 2010.
- [12] Jespersen D C. Acceleration of a CFD code with a GPU [R]. NASA Advanced Supercomputing Technical Report, NAS-09-003, 2009.
- [13] Brandvik T, Pullan G. An accelerated 3D Navier-Stokes solver for flows in turbomachines[J]. Journal of Turbomachinery, 2011, 133(2): 021025. 1-021025. 9.
- [14] 王健, 许明, 葛蔚, 等. 单相流动数值模拟的 SIMPLE 算法在 GPU 上的实现 [J]. 科学通报, 2010, 55(20): 1979-1986.
WANG Jian, XU Ming, GE Wei, et al. GPU accelerated direct numerical simulation with SIMPLE arithmetic for single-phase flow[J]. Chinese Science Bulletin, 2010, 55(20): 1979-1986. (in Chinese)
- [15] 董延星, 李新亮, 李森, 等. GPU 上计算流体力学的加速 [J]. 计算机系统应用, 2011, 20(1): 104-109.
DONG Yanxing, LI Xinliang, LI Sen, et al. Acceleration of computational fluid dynamics codes on GPU[J]. Computer Systems and Applications, 2011, 20(1): 104-109. (in Chinese)
- [16] 张兵, 韩景龙. 一种 GPU 上的 CFD 并行计算方法[C]//第十一届全国空气弹性学术交流会. 昆明: 中国空气动力学学会, 中国力学学会, 2009: 371-377.
- [17] 张兵, 韩景龙. 基于 GPU 和隐式格式的 CFD 并行计算方法[J]. 航空学报, 2010, 31(2): 249-256.
ZHANG Bing, HAN Jinglong. Parallel computing methods for CFD using a GPU and implicit scheme[J]. Acta Aeronautica et Astronautica Sinica, 2010, 31(2): 249-256. (in Chinese)
- [18] 苗树明. N-S 方程在 GPU 上的实现[D]. 上海: 上海交通大学, 2011.
MIAO Shuming. Implementation of N-S equations in parallel on GPU[D]. Shanghai: Shanghai Jiaotong University, 2011. (in Chinese)
- [19] 宁方飞. 考虑真实几何复杂性的跨音压气机内部流动的数值模拟[D]. 北京: 北京航空航天大学, 2002.
NING Fangfei. Numerical investigations of flows in transonic compressors with real geometrical complexities[D]. Beijing: Beijing University of Aeronautics and Astronautics, 2002. (in Chinese)
- [20] NVIDIA Corporation. NVIDIA CUDA C programming guide, version 4. 2[EB/OL]. [2012-11-06]. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
- [21] Jameson A, Yoon S. Lower-upper implicit schemes with multiple grids for the Euler equations[J]. AIAA Journal, 1987, 25(7): 929-935.
- [22] Blazek J. Computational fluid dynamics: principles and applications[M]. New York: Elsevier Science, 2001.
- [23] NVIDIA Corporation. NVIDIA'S next generation CUDA™ compute architecture: KEPLER™ GK110 [EB/OL]. [2013-06-04]. <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-White-paper.pdf>.