GPU 上计算流体力学的加速①

董廷星[1,2]，李新亮[3]，李 森[1,2]，迟学斌[1]

[1] (中国科学院 计算机网络信息中心，北京 100190)
[2] (中国科学院 研究生院，北京 100190)
[3] (中国科学院 力学研究所，北京 100180)

摘 要：本文将计算流体力学中的可压缩的纳维叶－斯托克斯(Navier-Stokes)，不可压缩的 Navier-Stokes 和欧拉（Euler）方程移植到 NVIDIA GPU 上。模拟了 3 个测试例子，2 维的黎曼问题，方腔流问题和 RAE2822 型的机翼绕流。相比于 CPU，我们在 GPU 平台上最高得到了 33.2 倍的加速比。为了最大程度提高代码的性能，针对 GPU 平台上探索了几种优化策略。和 CPU 以及实验结果对比表明，利用计算流体力学在 GPU 平台上能够得到预想的结果，具有很好的应用前景。

关键词：GPU 计算；CUDA；计算流体力学

Acceleration of Computational Fluid Dynamics Codes on GPU

DONG Ting-Xing[1,2], LI Xin-Liang[3], LI Sen[1,2], CHI Xue-Bin[1]

[1] (Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China)
[2] (Graduate University of Chinese Academy of Sciences, Beijing 100190, China)
[3] (LHD, Institute of Mechanics, CAS, Beijing 100180, China)

Abstract: Computational Fluid Dynamic (CFD) codes based on incompressible Navier-Stokes, compressible Euler and compressible Navier-Stokes solvers are ported on NVIDIA GPU. As validation test, we have simulated a two-dimension cavity flow, Riemann problem and a transonic flow over a RAE2822 airfoil. Maximum 33.2x speedup is reported in our test. To maximum the GPU code performance, we also explore a number of GPU-specific optimization strategies. It demonstrates GPU code gives the expected results compared CPU code and experimental result and GPU computing has good compatibility and bright future.

Keywords: GPU computing; CUDA; CFD

1 引言

近年来，图形处理器(GPU)的取得了飞速发展。典型的如 NVIDIA?公司 Tesla 系列 GPU 已经能达到 1Teraflop/s 的性能[1]。为了实现 GPU 上的通用计算，先后涌现了若干编程模型如斯坦福大学的 Brook[2]，以及后来的 Brook+和 OpenCL[3]。但最具代表性的是 NVIDIA?公司推出的统一计算设备架构(CUDA)。CUDA 建立在 C 语言的最小拓展级上，使得编程人员不再需要掌握复杂的图形知识从而让 GPU 编程变得简单易学。CUDA 推出之后，已经在计算流体力学(CFD)上得到了广泛应用[4-7]。本文以计算流体力学中的三个问题为例子，在 GPU 平台上实现 Navier-Stokes(N-S)方程和欧拉方程的求解。

文章的结构如下：我们首先在第一章简单介绍下 NVIDIA CPU 和 CUDA 的编程模型。接着给出三个 CFD 测试例子的描述。在第三章我们详细阐述了欧拉方程在 CPU 的应用。第四章，我们提出了几种优化策略来进行性能优化。最后我们给出 CPU 平台上 Fortran 代码和 GPU 平台上 CUDA 代码结果的对比。

① 基金项目:中国科学院知识创新工程青年人才领域项目(0815011103)
收稿时间:2009-04-29;收到修改稿时间:2010-05-20

2 NVIDIA GPU与CUDA编程模型

目前为止，只有 NVIDA GPU 支持 CUDA 编程。NVDIA 和 AMD (ATI)的 GPU 都支持 OpenCL[8]. 但是目前后者不如前者的执行效率高。因此我们这里采用 CUDA 作为编程平台。以 NVIDIA 的 TeslaC1060 为例，C1060 是一款专门为科学计算而设计的处理器[1]，并不支持图像处理功能，因此已经脱离传统意义上显卡概念。Tesla 采用众核设计，包含 30 个流多处理器(Streaming Multiprocessors,SM)。每个 SM 包含 8 个标量处理器(Scalar Processor, SP).一个 SM 共享 16KB 的共享存储器(Shared Memory)和 8192 个寄存器。每个 SP 一个时钟周期可以进行一个乘法和加乘操作，于是 TeslaC1060 的理论峰值的浮点性能为 933GFLOP/s[1]。相比之下，Intel 高端服务器至强系列中的 E5450 和 X5570 的理论峰值的浮点性能分别只有 48 GFLOP/s 和 46.88 GFLOP/s。因此 GPU 与 CPU 相比，在计算能力上有一个巨大的飞跃。Tesla C1060 可拓展显存为 4 GB[1]。带宽上，C1060 最高带宽可达 102 GB/s，差不多比 X5570 和 E5450 的 10.6GB/s 高一个数量级[1]。

CUDA 采用普通编程人员所熟悉的 C 语言的编程风格，因此 GPU 编程简单易学[9]。与 C 语言中用函数封装类似，CUDA 以内核函数(kernel)的形式来执行。在 kernel 函数中，GPU 上启动众多的线程(threads)来执行。线程是执行的最小单位，这和 CPU 中的线程的概念类似。但是比后者更加灵活，而且线程间的切换和调度为零开销[4]。thread 进一步被组织成线程块(block)和网格(Grid)的形式，如图 1。

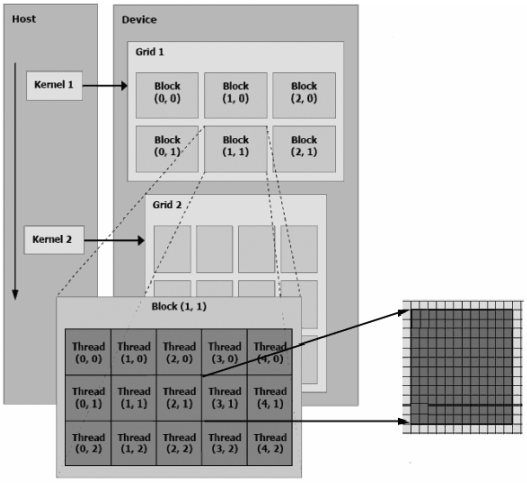

图 1 CUDA 线程层次；ghost cell (浅色)和内部单元(深色)，block 边界的 thread 需要访问 ghost cell

同一个 block 中的 thread 执行在同一个 SM 上，这保证了同一个 block 的 thread 之间能够通过读取 shared memory 通信和进行同步。每个 block 必须配置成相同大小，并且 block 进一步组织成 grid，因此 grid 中的总 thread 数目等于 block 内的 thread 数与 block 数目的乘积。由于目前的 GPU 上还没有配备对显存端全局存储器(global memory)的缓存(cache),因此需要启动足够多的 thread 数来计算以隐藏读取内存的延迟。CUDA 是一种高度并行化的解决方案，编程者需要依据 CUDA 的线程层次，把一个大规模的任务划分成众多可以并行的小规模任务来执行。

3 测试例子与数值方法

Navier-Stokes(N-S)方程是描述流体运动规律的基本方程。在不同的物理情况下，N-S 方程有其特殊形式。这里我们以 2 维 RAE2822 的机翼绕流模拟为例，讨论描述其流动特征的可压缩的 N-S 方程[10]。

$$\frac{\partial}{\partial t}U+\frac{\partial}{\partial x}f_1+\frac{\partial}{\partial y}f_2+\frac{\partial}{\partial z}f_3=\frac{\partial}{\partial x}V_1+\frac{\partial}{\partial y}V_2+\frac{\partial}{\partial z}V_3$$

其中，

$$U=[\rho,\rho u,\rho v,\rho w,E]^T$$

$$f_1=[\rho u,\rho u^2+p,\rho uv,\rho uw,u(E+p)]^T$$

$$f_2=[\rho v,\rho vu,\rho v^2+p,\rho vw,v(E+p)]^T$$

$$f_3=[\rho w,\rho wu,\rho wv,\rho w^2+p,w(E+p)]^T$$

$$V_1=[0,\sigma_{11},\sigma_{21},\sigma_{31},u\sigma_{11}+v\sigma_{21}+w\sigma_{31}+k\frac{\partial T}{\partial x}]^T$$

$$V_2=[0,\sigma_{12},\sigma_{22},\sigma_{32},u\sigma_{12}+v\sigma_{22}+w\sigma_{32}+k\frac{\partial T}{\partial y}]^T$$

$$V_3=[0,\sigma_{13},\sigma_{23},\sigma_{33},u\sigma_{13}+v\sigma_{23}+w\sigma_{33}+k\frac{\partial T}{\partial z}]^T$$

ρ,u,v,w,E 分别为密度，三个坐标方向速度和能量。$\sigma_{i,j},k,T$ 分别为粘性应力张量分量，热传导系数和温度。

方程组中，第一个方程为连续方程，第 2，3，4 为 3 个坐标方向的动量方程。最后一个为能量方程。方程的右端对应粘性项，左端对空间的偏导数项为非粘性项。当流场中粘性可以忽略不计的时候，N-S 方程右端的粘性项消失，N-S 方程退化为欧拉（Euler）方程。

对于可压缩的 N-S 方程，粘性项首先进行矢量分裂为正负通量。正通量采用 5 阶迎风(后差)格式，负通

量采用 5 阶迎风(前差)格式，如公式(1)，(2)。方程右端的粘性项采用 6 阶精度的中心差分格式，如公式(3)[10]。

$$f_j^+ = (-2f_{j-3} + 15f_{j-2} - 60f_{j-1} + 20f_j + 30f_{j+1} - 3f_{j+2})/60h \quad (1)$$

$$f_j^- = -(-2f_{j+3} + 15f_{j+2} - 60f_{j+1} + 20f_j + 30f_{j-1} - 3f_{j-2})/60h \quad (2)$$

$$V_j = (45(V_{j+1} - V_{j-1}) - 9(V_{j+2} - V_{j-1}) + (V_{j+3} - V_{j-3}))/60h \quad (3)$$

其中，下标 j 代表网格节点，h 为空间步长。

为了能够更加广泛的检验计算流体力学在 GPU 上的适用性，我们实现了三个计算流体力学的例子，分别将不可压缩的 Navier-Stokes，可压缩的 Navier-Stokes 方程和欧拉方程移植到 GPU。三个例子的描述见表 1。

表 1　测试例子

测试名称	方程	数值方法	计算规模
2 维方腔流[11]	不可压缩的 N-S	3 阶精度的 Quick 格式;	96*96- 1024*1024
2 维黎曼方法[12]	欧拉方程	2 阶精度的 NND 格式;	512*512-1024*1024
2 维 RAE2822 机翼绕流[13]	可压缩的 N-S	非粘性项目 5 阶迎风格式; 粘性项 6 阶中心格式；时间 3 阶 Runge-Kutta;	369*65-1024*128

4　CFD代码在CUDA上的实现

4.1 执行流程

GPU 上无法运行操作系统，而是作为 CPU 的协处理器来运行。在计算中 CPU 和 GPU 承担不同的任务，如图 2。

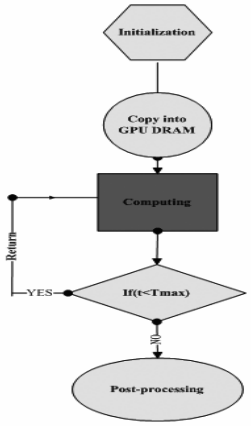

图 2　计算流程图

由于牵涉到对硬盘的 I/O 操作，CPU 负责数据的初始化。然后数据拷入到 GPU 的显存空间。GPU 承担计算最密集的部分(深色)，直至循环结束，GPU 计算退出。最后数据从显存拷贝到主存中，由 CPU 进行数据的后处理。

4.2 欧拉方程在 CUDA 上的实现

我们以 Euler 方程的 5 点差分为例来描述使用有限差分的 2 维计算流体问题如何在 CUDA 中实现。

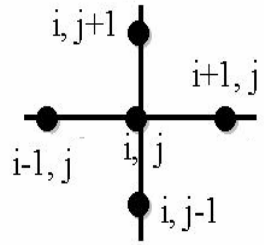

图 3　点差分格式

(1) 区域分解和线程映射。如前所述，采用类 C 的编程风格，CUDA 程序写成 kernel 函数的形式。在 kernel 函数中成千上万的线程并行或者并发的执行。对于上述三个计算区域为结构化网格的例子，一个自然的方法就是将首先求解区划分成适合线程块(block)的小区域。block 中，一个 thread 计算对应小区域的一个元素(element)，并用一个唯一的 ID 来标识。比如计算规模为 256*256，自然的线程映射则在线程网格中安排 256*256 个线程，每个线程一一对应计算区域中的一个元素。同时 block 的大小也需要指定，我们在本文中设置包含 16*16 个 threads 的 block(设计为 16 的倍数是为了保证后面提到的性能攸关的 global memory 的合并访问)。但在机翼模拟中计算区域为 369*65，不是 16 的倍数，我们的策略是：线程网格 grid 设计为 384*80(总的线程数高于 365*65)，block 为 16*16，一个线程依旧计算一个区域中的元素，但是多余的线程并不真正参与计算。

(2) 内存读取。算法通常决定了格式的复杂度。考虑到一个 2D 的 5 点格式的差分(比如中心差分格式)如图 3，为了更新元素 i, j, 线程 i, j 需要读取周围的四个邻元素。这意味着每个元素至少要读取 5 次。显然，直接从显存中的全局存储器(global memory)中访问数据代价太高了。因为 global memory 的读取速度为 500 时钟周期(cycle)的延迟。一个有效的方式是将一个线程块 block 的数据一次

性拷入到共享存储器 shared memory 中, 以 shared memory 为缓存，线程从 shared memory 中读取数据，因为 shared memory 是片上内存，读取速度只有一个 cycle 的延迟。因而对于频繁使用的数据，使用 shared memory 大大减少 global memory 冗余访问，加快了内存的读取速度。对于有限差分等数据具有良好的局部性的算法，使用 shared memory 通常能够达到提高带宽的目的[4]。

(3) 拓展虚拟网格。仍然以 5 点格式的差分为例，在线程块 block 边界上的元素只有三个邻元素，因而需要拓展虚拟单元(ghost cell)来存储 block 边界线程需要的数据。在 5 点格式差分中 ghost cell 需要 1 层。Ghost cell 的概念和 MPI 中的 ghost cell 概念类似，不同的是 CUDA 中的不同 block 中的线程无法像 MPI 中的进程一样通信和同步。

(4) 执行。线程网格中，每个线程并行或者并发的执行。同一个 block 中的线程可以通过共享的 shared memory 和显存通信和同步。但是各个 block 是独立执行的，执行顺序由软件控制，编程人员无从控制和了解。只有当 kernel 函数执行完毕后，所有的 block 才可以进行同步。

(5) 解的推进。方程的解随时间步的递进而更新。程序中 CUDA 的主体部分即时间步推进构成的循环(loop)。在 loop 中 kernel 函数反复迭代来更新方程的解，直到计算时间溢出。大多数情况下，loop 由多个 kernel 函数构成(kernel 的多少通常和算法复杂程度有关)。

4.3 性能优化

程序的关键部分为包含 kernel 函数反复迭代的循环，因而是影响性能的热点(hotspot)。针对 CUDA 特性，我们探索了几种优化策略。

（1）最大化全局存储器的合并访问。当连续的 16 个线程同时进行 global memory 访问，如果内存地址满足对齐要求，将合并为一个访问事务，否则各个线程将启动独立的访问请求[3]。非合并访问将大大降低内存带宽，损失性能，在 CUDA 程序的优化中，首先考虑的就是最大化合并访问比重，减少非合并访问。为了满足地址的对齐要求，线程必须访问连续的内存地址。在 5 点差分格式中，如果线程直接读取 global memory，内存的读取模式很显然是非合并的。因为除了中心的元素，线程还必须读取周围四个邻元素，从而导致线程对地址的交叉访问。在我们的程序中，block 设计为 16*16，我们利用 shared memory 作为数据的“中转站”大大降低了非合并访问问题。因为从 global memory 到 shared memory，线程连续访问地址，满足合并访问的要求。但是对于 ghost cell 的读取，不满足合并访问，因为 block 边界处的线程必须单独去读取 ghost cell。对 5 点差分格式，由于 ghost cell 只有 1 层，非合并访问的代价不大(尽管无法完全消除)。

（2）使用纹理存储(texture memory)。但对于高阶精度的格式，由于差分格式更加复杂, 比如机翼模拟中使用的 6 阶精度的中心差分格式，计算一个元素，需要访问周围 6 个邻元素。 因而需要拓展 3 层的 ghost cell，导致 ghost cell 比重增大。比如对于 16*16 的 block, ghost cell 和内部单元的比例为 12:16(3:4)，将导致非常严重的非合并访问。为了减少大比重的 ghost cell 访问带来的性能损失，对于高阶精度复杂的差分。我们使用 texture memory 来加速内存读取。Texture memory 是在显存中的一块只读空间，进行了二级缓存。纹理存储不受制于如全局存储器下的合并访问的限制，在通过 texture memory 来读取的时候，附近的元素(称为 pixels, 像元)将自动读取，因而很适合数据局部性强的内存读取模式。当 global memory 因非合并访问而造成带宽限制时，使用 texture memory 可以表现为更高的带宽。但注意应尽量实现合并访问，因为只有后者才能带来内存带宽的最大化[3]。而且纹理存储的缓存空间是有限的，在计算能力 1.0 下，每个多处理器最多有 8KB 的缓存[9]。

（3）降低分支。Tesla 使用一种叫做单数据多线程的执行模型(single-instruction, multiple-threads，SIMT)。每个 SM 的 SIMT 单元以连续的一个线程块 block 中的 32 个线程(称为一个 warp)为单位来创建，管理，切换和执行指令[9]。一个 warp 中，32 个线程从同一程序地址启动，执行相同的指令。如果 warp 中线程出现条件分支而分散，那么 waro 所有的线程将依次串行执行所有的分支路径，这将大大降低指令的吞吐量。为避免同一个 warp 中出现分支，block 的大小最好设置为 32 个倍数。

（4）最小化 GPU 和 CPU 之间的数据传输。因为目前连接 GPU 和 CPU 的 PCI-Express 总线的带宽只有 8GB/s，大大落后于 GPU 和 CPU 的带宽。数据多次在显存与主存中传输将导致 PCI-E 瓶颈。因此最好将中间的计算过程全部放入 GPU 中执行，让 GPU 和 CPU 之间的数据交换只发生在计算前后。在本文中的计算中，GPU 全部承担中间部分的计算中，仅在前后端与 CPU 之间发生数据交换。

5　结果与对比

我们实现了三个计算流体力学的例子来验证 GPU 模拟并测试 GPU 对 CPU 的加速比。测试平台如下：

CPU 平台：Intel 至强 E5450 主频 3.00GHz 和 至强 X5570 主频 2.93GHz，Intel 的 Fortran 和 C 编译器，优化选项为 –O3。

GPU 平台: Tesla C1060，NVCC 编译器，CUDA 版本 2.2。

我们分别在 X5570 和 Tesla C1060 上实现了 2D 的黎曼问题的模拟(测试 2)。计算时间如表 2。由表中可知，在计算规模为 512*512 时，加速比为 20.9 倍，随着规模进一步扩大到 1024*1024，加速比达到 33.2 倍。

对于 2D 的方腔流问题，我们在 GPU 的计算中采用了双精度，在 GPU 上，双精度的计算能力为 78 GFLOP/s 要远远小于单精度的 933 GFLOP/s。双精度的加速比不如黎曼问题明显(黎曼问题使用单精度)。

图 4 为方腔流的流线图。图 5 显示了 t=0.1 时刻，2D 黎曼问题的密度分布。这些定性的结果显示 GPU 计算给出了合理的计算结果。图 6 显示了 x 方向中线上的速度 u 的图，定量的结果显示 GPU 和 CPU 的计算结果仍然有微小的差异。如何解释在 GPU 与 CPU 上的差异仍然需要我们进一步的研究。

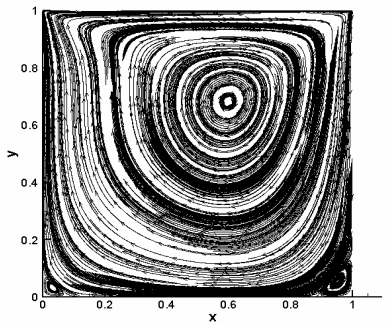

图 4　雷诺数为 100，网格为 256*256 时候方腔流的流线图

图 7 为机翼模拟中翼体上的压力系数。CPU/GPU 和实际的实验结果[15]对比发现吻合的很好。而且在消除机翼头部激波的震荡上 GPU 的结果比 CPU 更好。

表 3 列出来机翼模拟中使用 8 个进程的 MPI 的 Fortran 代码和 CUDA 代码的时间对比。可以看出在较大的计算量时，GPU 更能展示其相对优势。当计算规模从 369*65 增大到 1024*128 时，计算量大约扩大了 5.5 倍。而 CPU 上墙上时间增长了 10 倍，但是 GPU 的计算时间仅仅增长了 1.5 倍。我们认为原因在于，当计算规模很局限的时候，GPU 中没有足够的线程在计算以掩盖对显存读取的延迟(对于 369*65 的规模平均每个 SM 上分配 4 个 block)众多的线程由于进行数据等待而闲置了 SM，限制了 GPU 计算能力的发挥，而当计算规模增大时，有足够的线程的计算以和内存读取叠加，GPU 的计算潜力得到了释放。CPU 程序方面，我们使用了最新 11 版本的 Intel Fortran 编译器，并打开了最高级优化选项-O3。对于两个计算规模下单个进程的数据大小都限制在了二级缓存之内。因此我们认为所得到的加速比是合理的。

表 2　2D 黎曼问题和 2D 方腔流问题的在 GPU 和 CPU 上的时间对比

测试名称	计算规模	GPU 时间	CPU 时间	GPU 加速比
2 维黎曼问题	512*512	4.67s	98s	20.9x
2 维黎曼问题	1024*1024	24.3s	806.5x	33.2x
2 维方腔流	256*256	6m2s	17m7s	3.0x

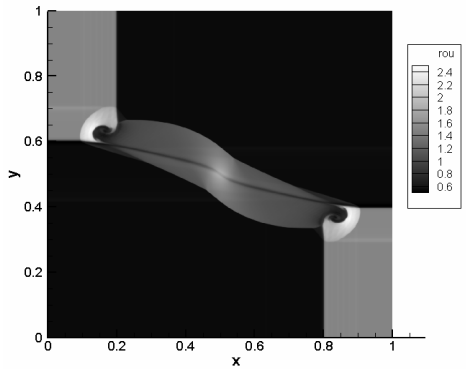

图 5　t=0.1 时，2D 黎曼问题的密度分布

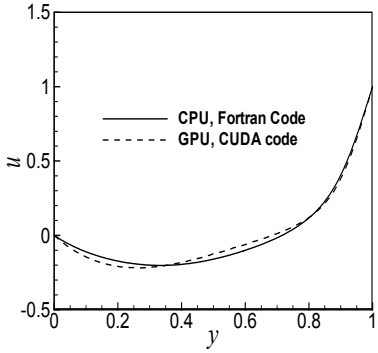

图 6　中线上（x=0.5）的速度 u

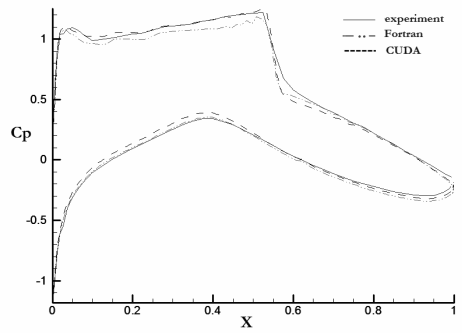

图 7 翼体表面的压力系数分布

6 结束语

我们使用图形处理器来加速计算流体力学代码，最高得到了 33.2 倍的加速效果，展示了 GPU 强大的计算潜力。我们相信，GPU 未来将成为科学计算领域的一个强大工具。但是在实际的很多应用中是需要双精度的，而在现有的架构下，双精度的计算能力还不够突出。最近 NVIDIA 宣布了下一代的计算架构 GT300，被认为是一个真正的 GPU 计算平台。不仅拥有更强大的双精度浮点能力而且对全局存储器添加了缓存。我们希望将我们的代码能够在新的平台上测试，并期待更强的性能提高。

表 3 10,000 步中 2D 机翼绕流模拟的时间

	1* C1060			8* E5450	
规模	Block 数量	每个SM上的block数	GPU 时间	墙上时间	加速比
369*65	120	4	3.8min	1.2min	0.31x
1024*128	512	17	6min	14min	2.33x

参考文献

1 Lindholm E, Nickolls J, Oberman S, Montrym J. Nvidia Tesla: A United Graphics and Computing Architecture. IEEE Computer Society.

2 Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream Computing on Graphics Hardware. SIGGRAPH 2004.

3 http://www.opencl.org/. [2010-05-19]

4 Brandvik T, Pullan G. An Accelerated 3D Navier-Stokes Solver for Flows in Turbomachines, Proceedings of GT2009 ASME Turbo Expo 2009: Power for Land, Sea and Air June 8-12, 2009, Orlando, USA.

5 陈飞国,葛蔚,李静海.复杂多相流动分子动力学模拟在 GPU 上的实现. Science in China, Serial B, 2008.

6 Andrew C, Fernando C, Rainald L, John W. 19th AIAA Computational Fluid Dynamics, June 22-25, San Antonio, Texas.

7 Jonathan MC, M. Jeroen M, A Fast Double Precision CFD Code using CUDA. IGPP UCLA, Los Angeles, CA 90095, USA.

8 [2010-05-19]http://ati.amd.com/technology/streamcomputing/op encl.html.

9 [2010-05-19] NVIDIA CUDA Compute Unified Device Architecture. Programming Guide.http://www.developer.download.nvidia.com.

10 李新亮,傅德薰,马延文.复杂流动直接数值模拟软件 OpenCFD.中国科学院力学研究所, 2008.

11 Zhang DL. Computational Fluid Dynamics, 2008.

12 Alexander K, Eitan T. Solution of Two-Dimensional Riemann Problems for Gas Dynamics without Riemann Problem Solvers. Wiley Periodicals, Inc. Numerical Methods Partial Differential Eq 18:584–608, 2002.

13 Cook PH, McDonald MA, Firmin MCP. Aerofoil RAE 2822 -Pressure Distributions, and Boundary Layer and Wake Measurements, Experimental Data Base for Computer Program Assessment. AGARD Report AR 138, 1979.

14 Elsen E, LeGresley P, Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU. Journal of Computational Physics, 2008.

15 [2010-05-19]http://www.grc.nasa.gov/WWW/wind/valid/raetaf/raetaf.html.