科学通报 2011 年 第 56 卷 第 28-29 期：2434 ~ 2444
论 文 www.scichina.com csb.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS

基于 CUDA 的格子 Boltzmann 方法：算法设计与程序优化

黄昌盛[①], 张文欢[②], 侯志敏[①], 陈俊辉[①], 李明晶[①], 何南忠[①*], 施保昌[①②]

① 华中科技大学数学与统计学院, 武汉 430074;
② 华中科技大学煤燃烧国家重点实验室, 武汉 430074
* 联系人, E-mail: nzhe@163.com

2011-05-30 收稿, 2011-08-12 接受
国家自然科学基金(51006039, 51006040)、国家重点基础研究发展计划(2011CB707305)、中央高校基本科研业务费专项基金(2010MS131, 2010QN057)资助项目

摘要 格子 Boltzmann 方法(LBM)由于其具有计算简单, 天然并行, 易于程序实现, 易于处理复杂边界等优点而成为流体建模和模拟的一种重要方法. LBM 的上述优点也使得其非常适合利用图形处理单元(graphic processing unit, GPU)进行大规模流体计算. 基于 GPU 的 CUDA(compute unified device architecture)编程平台, 首先设计了相应的 LBM 算法, 并以二维方腔流、二维圆柱绕流以及三维方腔流为例, 着重探讨了存储器访问优化等优化技术的作用; 此外, 本文也对程序的性能进行了详细分析. 结果表明, 本文的算法取得了理想的加速效果, 证实了 GPU 与 LBM 的良好匹配关系.

关键词
格子 Boltzmann 方法
CUDA
并行计算
GPU
优化

近半个多世纪以来, 随着科学技术水平的不断提高, 人们需要处理的科学及工程问题也越来越复杂. 面对这些问题, 传统的实验和理论方法越来越难以满足实际需要, 时代呼吁新的科学手段的出现. 20 世纪 50 年代计算机的诞生使这种设想成为可能. 随着计算机和计算方法的飞速发展, 一种以计算方法为基础、运用计算机对科学及工程实际问题进行数值模拟的科学手段得以广泛地应用. 科学计算或曰计算机模拟也逐渐成为与传统的科学方法—理论和实验相并列的“第三种科学方法”, 其发展水平也越来越成为某一国家在现代科学和高技术研究与开发中竞争能力的重要标志. 一般来说, 这种发展水平的高低主要取决于两个主要因素, 即高性能计算工具(计算机)和高效计算方法. 前者是科学计算的硬件, 后者是科学计算的软件, 两者具有同等重要的地位. 因此, 科学计算的发展也基本按照这两条主线不断发展.

对于高性能计算工具, 处理器扮演最为关键的角色. 在过去的 10 年中, CPU 设计者受功率的限制不再在单一 CPU 上添加更多的晶体管, 而是把主要精力放在发展多核 CPU 体系结构上. 另一方面, 由于图形处理本身就是一个并行任务, 采用流处理体系结构的 GPU 更容易采用多核策略, 且更适合于并行计算. 因此, GPU 的通用计算能力越来越受到关注, 基于 GPU 的并行计算硬件设计逐渐成为高性能计算领域的新宠, 代表了高性能计算硬件发展的最新趋势. NVIDIA 公司无疑是这一领域的领导者, 其 2007 年以来推出的 GPU 并行计算开发环境 CUDA(compute unified device architecture)则将这一领域的水平推向了一个前所未有的高度. 与此同时, 作为高效计算方法的重要代表, 格子 Boltzmann 方法(lattice Boltzmann method, LBM)自产生到现在 20 余年间受到了计算流体力学等诸多领域内学者的普遍青睐. 与传统的建立在连续介质模型上的计算流体力学(computational fluid dynamics, CFD)相比, 该方法是 20 世纪 80 年代

英文引用格式: Huang C S, Zhang W H, Hou Z M, et al. CUDA based lattice Boltzmann method: Algorithm design and program optimization (in Chinese). Chinese Sci Bull (Chinese Ver), 2011, 56: 2434–2444, doi: 10.1360/972011-1078

中期发展起来的一种流体建模与模拟的新方法，它不仅具有算法简单、易于处理复杂边界条件、压力能够直接求解、编程容易等优势，而且还具有天然的并行性，这些优势使得 LBM 非常适合进行大规模的并行计算.

因此，自 GPU 开始用于通用计算的那一刻，国内外就不断有研究者试图将 GPU 与 LB 方法结合起来以便能够高效处理大规模流体计算问题. 这种尝试按是否应用 CUDA 环境主要分为两大类，一类是基于 GPGPU，另一类则是基于 CUDA. 前者直接将把基本的计算任务映射为对纹理的渲染过程，Boltzmann 方程的演化完全通过栅格化和帧缓冲操作来实现. 对比 CPU 上的实现，这种实现可以达到至少一个数量级的加速比[1]，相关应用包括实时油墨在吸水纸上分散[2]，城市安全的色散仿真及可视化[3]，肥皂泡的模拟[4]，互溶二元混合物的模拟[5]，多相流环境下的熔化和流动[6]，热气晃动和幻影的可视化模拟[7]，实时草波动[8]，三维实时流体模拟[9]，混合流模拟[10]等. 然而，这些应用多为图形应用，使用的都是接近于硬件的编程模式，不易为普通程序员所掌握且加速比也十分有限；相比基于 GPGPU 实现，直接利用 CUDA 编程平台实现 LB 计算，编程过程简单，易于为普通程序员所理解. 因此，尽管CUDA推出时间较晚，仍有不少学者对此进行了较深入的研究，如 Tölke 等人[11,12]使用 CUDA 实现了二维多孔介质和三维绕流 LBM 代码，Kuznik 等人[13,14]实现了二维和三维方腔流的代码；Bernaschi 等人[15]实现了多组分的 LBM 模型等. 对比 CPU 上的实现，这种实现一般可以达到 50 以上的加速比. 除此之外，李博等人[16]还实现了一种耦合 Nvidia 和 AMD 的两类 GPU 的 LBM 凹槽流模拟的算法，所能带来的加速比甚至到 229.83.

从上述分析不难看出，目前基于GPU的LB程序实现已经取得了相当可观的成绩，初步显示了 GPU 与 LB 方法的良好匹配关系. 尽管如此，已有文献所显示的基于 GPU 的 LB 程序能达到的加速比仍然十分有限，程序性能有待进一步提高，许多方面仍缺乏相对系统地探讨，如 LB 程序的算法优化，GPU 优化等. 鉴于此，本文拟以二维方腔流、二维圆柱绕流以及三维方腔流为例，实现相应的 CUDA 环境下的 LB 模拟程序，着重探讨了算法优化，CUDA 程序优化，存储器访问优化、资源均衡等优化手段在程序设计中的作用，并利用程序的一系列性能指标如格子更新率、加速比等进行详细分析.

本文将对 GPU 和 CUDA 进行一个简要介绍；将对格子 Boltzmann 方法以及用到的基本模型予以说明；以简单的二维方腔流为例，详细介绍其 LB 程序的 CUDA 实现过程并分析不同实现方法在程序性能上的差异，探讨算法优化以及 GPU 程序优化在程序设计中的应用；将以更复杂的二维圆柱绕流和三维方腔流为例，在继续探讨类似议题的同时，讨论物理问题的复杂边界条件以及维数变化对程序性能的影响.

1 GPU 及 CUDA

1.1 GPU 的发展

NVIDIA 公司 1999 年推出了全球第一颗 GPU——GeForce256. 此后，NVIDIA 公司相继推出第一款拥有可编程顶点着色器的 GPU——GeForce3 以及第一款使用 32 位浮点流水线作为可编程的顶点处理器的 GPU——GeForce Fx，2006 年的 GeForce 8 系列 GPU 率先采用了统一渲染架构，以通用渲染单元替代了原来分离的着色单元和像素着色单元，能够支持 DirecrtX 10.0. 此后，CUDA 正式发布，引发了 GPU 通用计算的革命[17].

GPU 在处理能力和存储带宽上相对于 CPU 有明显优势，GPU 可以通过增加并行处理单元和存储器控制单元的方式提高处理能力和存储器带宽. CPU 和 GPU 的浮点计算能力差异大的主要原因是：GPU 是专为密集运算、高度并行计算而设计，与 CPU 相比，GPU 内部更多的晶体管用于数据处理而不是数据缓存或控制.

1.2 CUDA 介绍

CUDA 是由 NVIDIA 推出的通用并行计算架构，可以使用 GPU 来解决商业、工业以及科学方面的复杂计算问题. CUDA 采用类 C 语言作为编程语言提供了大量的高性能计算指令开发能力，使开发者能够在 GPU 的强大计算能力的基础上建立起一种效率更高的密集数据计算解决方案，所编写出的程序也就可以在支持 CUDA 的处理器上以超高性能运行.

CUDA 将 GPU 视为一个并行数据计算的设备，对所进行的计算进行分配和管理. 在 CUDA 的架构中，不需要像过去的 GPGPU 计算那样将计算映射到

图形 API(OpenGL 和 Direct 3D)中, 从而降低了 CUDA 的开发门槛.

2 LBM 基本理论

2.1 格子 Boltzmann 方法的基本原理

格子 Boltzmann 方法作为一种新的流体力学计算方法, 受到国内外学者的普遍关注. 从计算的角度看, LBM 具有如下的显著优势: 首先, 该方法是一种显式时间推进方法, 每个时间步的计算量为 O(*MN*)(*M* 为离散速度数, *N* 为计算格点数), 在计算效率方面高于一般的数值方法; 其次, LBM 的物理背景清晰且演化过程极为简单, 这使得其在处理复杂边界和程序设计方面均具有一定的优势; 最后, LBM 中涉及的计算都是局部性的, 因而, 该方法具有天然的并行性, 非常适合在大规模并行计算机上运行.

LBM 的演化方程可表示为如下形式:

$$f_i(\boldsymbol{x}+c\boldsymbol{e}_i\Delta t, t+\Delta t) = f_i(\boldsymbol{x},t) + \Omega_i(f_i, f_i^{(eq)}),$$

其中 $f_i, f_i^{(eq)}$ 分别为沿方向 $\boldsymbol{e}_i$ 的粒子分布函数和局部平衡态分布函数, $c = \Delta x/\Delta t$ 为粒子速度, Ω_i 为碰撞算子. 通过引入 Bhatnagar-Gross-Krook(BGK)碰撞算子, 我们可以得到如下的 lattice BGK (LBGK) 模型:

$$f_i(\boldsymbol{x}+c\boldsymbol{e}_i\Delta t, t+\Delta t) = f_i(\boldsymbol{x},t) - \frac{1}{\tau}\left[f_i(\boldsymbol{x},t) - f_i^{(eq)}(\boldsymbol{x},t)\right],$$

其中 τ 为无量纲松弛时间.

2.2 DdQq 和 iDdQq 模型

Qian 等人于 1992 年提出了著名的 DdQq 模型, 其平衡态分布函数可表示为

$$f_i^{(eq)} = \rho w_i\left(1 + \frac{c\boldsymbol{e}_i\cdot\boldsymbol{u}}{c_s^2} + \frac{(c\boldsymbol{e}_i\cdot\boldsymbol{u})^2}{2c_s^4} - \frac{\boldsymbol{u}\cdot\boldsymbol{u}}{2c_s^2}\right),$$

其中流体的宏观密度和速度分别由下式计算得到:

$$\rho = \sum_i f_i, \quad \rho\boldsymbol{u} = \sum_i c\boldsymbol{e}_i f_i.$$

较常用的模型有 D2Q9 模型, D3Q19 模型, D3Q15 模型, 具体的方向向量 $\boldsymbol{e}_i$, 权系数 w_i, 以及参数 c_s 在不同的模型中取值不同, 具体的取值可参见文献[18].

为了减少 DdQq 模型的压缩效应, 我们曾给出了一类新的模型[19], 该模型的平衡态分布函数具有如下形式:

$$g_i^{eq} = \lambda_i p + \omega_i\left[\frac{\boldsymbol{c}_i\cdot\boldsymbol{u}}{c_s^2} + \frac{(\boldsymbol{c}_i\cdot\boldsymbol{u})^2}{2c_s^4} - \frac{\boldsymbol{u}\cdot\boldsymbol{u}}{2c_s^2}\right],$$

其中 $\lambda_0 = \dfrac{\omega_0 - 1}{c_s^2}$, $\lambda_i = \dfrac{\omega_i}{c_s^2}$, $i \neq 0$, $\boldsymbol{c}_i = c\boldsymbol{e}_i$.

与 DdQq 模型相似, 该模型的演化方程可表示为

$$g_i(\boldsymbol{x}+c\boldsymbol{e}_i\Delta t, t+\Delta t) = g_i(\boldsymbol{x},t) - \frac{1}{\tau}\left[g_i(\boldsymbol{x},t) - g_i^{(eq)}(\boldsymbol{x},t)\right],$$

流体的宏观速度和压力分别定义为

$$\boldsymbol{u} = \sum_{i\neq 0}\boldsymbol{c}_i g_i, \; p = \frac{c_s^2}{1-\omega_0}\left(\sum_{i\neq 0} g_i - \omega_0\frac{\boldsymbol{u}\cdot\boldsymbol{u}}{2c_s^2}\right).$$

上述几个方程一起构成了一个新的不可压格子 Boltzmann 模型, 称之为 iDdQq 模型. 与 DdQq 模型相对应, 常用的 iDdQq 模型有 iD2Q9, iD3Q19, iD3Q15 模型, 模型中各种参数的取值参见文献[19]. iDdQq 模型不但可以有效降低可压缩效应带来的误差, 而且没有增加任何附加的计算量, 保持了原有格子 Boltzmann 方法的优点.

2.3 边界处理

LBM 的微观特点使得其在处理复杂边界时较传统方法更具优势. 边界条件的处理方法在 LBM 中起着重要的作用, 并且对模型的精度和稳定性都有很大的影响. 常用的有反弹格式、动力学格式、外推格式等, 但这些边界处理格式在稳定性与通用性方面受到了一定的限制. 为此, 本文采用一种计算简单、通用性强且数值稳定性好的边界处理方法: 非平衡态外推边界处理格式[20].

3 二维方腔流的数值模拟

在各种流动问题中, 方腔流以其简单的几何形状和丰富的漩涡结构, 一直作为测试和评估各类数值方法的基准问题. 图 1 给出了二维方腔流的示意图, 其中左右壁和下壁均为固壁, 上板拖动速度为 *U*.

3.1 算法设计

基于 CUDA 的程序计算基本步骤为: 首先, 分别在 GPU 和 CPU 上分配内存, 在 Host 端(CPU)初始化相关数据; 其次, 设置 grid 和 block(CUDA 的编程层次结构)的大小, 在 Device 端(GPU)进行并行计算; 最后, 当计算结果满足终止准则后, 将数据拷贝到 Host 端, 得到最终结果, 释放 GPU 和 CPU 的内存.

对于基于 CUDA 的 LBM 程序, 由于对每个格点的计算十分简单, 因而影响程序性能的瓶颈主要在

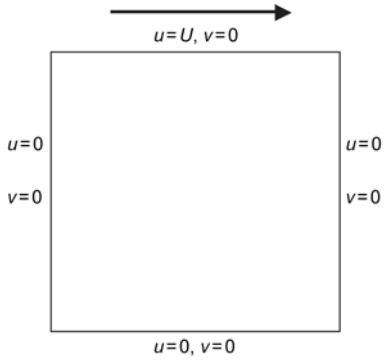

图 1　二维方腔流示意图

于数据的访存而非计算. 相关试验结果表明, 访存数据所耗费的时间占据总运行时间的 80%以上. 因此, 接下来我们将围绕数据访存开展算法设计与程序优化.

在程序中, 我们采用两个同样大小的一维数组来储存所有格点的分布函数, 分别设为 f,F. 假设网格规模为 $NY\times NX$, 坐标为(x,y)的节点沿 k 方向的分布函数存储在第$(k\times NX\times NY+y\times NX+x)$个位置. 对于奇数时间步, 数据从 f 中读取, 经碰撞流动后储存在 F 中; 对于偶数时间步则反过来执行. 为减少数据访存次数, 我们将碰撞与流动两个步骤在一个 kernel 中合并执行.

从程序设计的不同角度考虑, 我们设计了以下三种算法:

(1) 算法 S1: 为了进一步减少访存次数, 将边界处理与碰撞流动合并, 即在同一个 kernel 中执行, 进而, 所有的数据在一个时间步只需访存一次. 为了在一个 kernel 里实现所有步骤, 需要先将 block 所处理格点的分布函数读入共享内存做边界处理.

由于程序中采用了非平衡态外推的边界处理格式, 在处理边界点时必须访问其邻近内部节点的分布函数. 考虑到CUDA程序并行执行的特点, 且保证访问到正确的数据, 必须将边界点及其邻近节点处在同一个 Block 中. 对于二维方腔流来说, 需要满足 blockDim.$y\geqslant 2$(block 在 y 方向尺寸); 三维情形则要求 blockDim.$y\geqslant 2$, blockDim.$z\geqslant 2$(block 在 z 方向尺寸).

(2) 算法 S2: 由于数据均存储在全局内存中, 因此是否满足合并访问是影响程序性能的最主要因素. 算法 S2 的目的就在于实现合并访问. 对于数据读取的合并访问容易满足, 因此在数据存储时满足合并访问至关重要.

假设在数据读取时已满足合并访问. 对于 LBM 程序来说, 碰撞步骤仅涉及当前节点分布函数的简单计算, 而迁移步骤需要将计算后的数据进行移动, 对于需要向左或向右的移动的方向, 则需要进行处理以满足合并访问.

借助于共享内存, 向右方向的迁移按照如图 2(a)所示, 每个 block 最右端的粒子分布值移到最左端, 其余的粒子依次向右移动; 向左迁移的处理方式如图 2(b)所示.

经过上述处理之后, 在存储时就能够满足合并访问, 但每个 block 两端的数据没有正确迁移, 因此需要另用一个 kernel 来处理这些数据, 以使其迁移到正确位置, 该 kernel 被称为 exchange(), 其实现执行过程如图 3 所示.

由于 block 两端的数据需要额外处理, 因此 blockDim.x 越大, 需要额外处理的数据所占比例越小, 就越能体现出合并访问的优势. 基于上述考虑, 边界处理另增加一个 kernel 执行. 另外, 需要指出的是, 算法 S2 正是当前国内外学者普遍采用的算法[11,13,16].

(3) 算法 S3: 算法 S2 实现了合并访问, 但需要增加一个 kernel, 并且增加了对 block 左右两端的数据

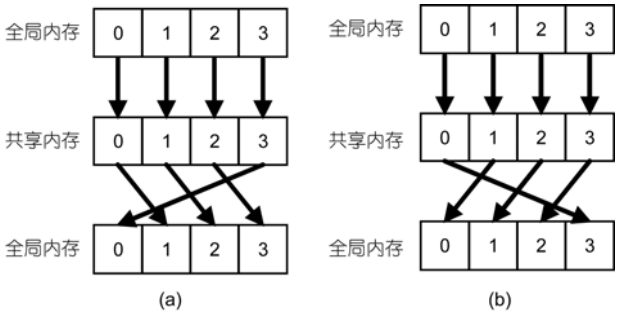

图 2　(a) S2 算法向右迁移的处理方式; (b) S2 算法向左迁移的处理方式

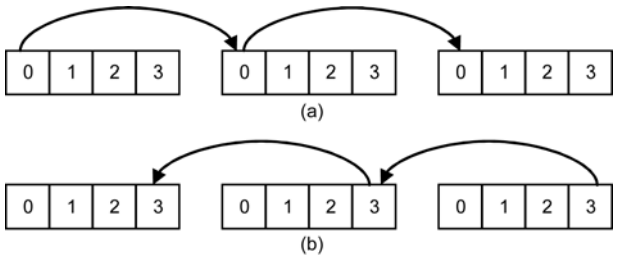

图 3　exchange()执行过程

(a) 向右迁移; (b) 向左迁移

的访问次数. 算法 S3 的目的在于不新增 kernel 的前提下实现合并访问, 它是将计算出的数据按照图 4 中的方案进行存储, 通过改变存储数据的线程, 除了两端的点外, 其他格点的数据均可满足合并访问.

3.2 程序优化

(ⅰ) 全局存储器访问优化. 对全局内存的访问是否满足合并访问是影响 CUDA 程序性能的主要因素之一. 由于数据主要存储于全局内存, 因此满足合并访问至关重要. 在算法设计中, 曾假定对数据的访问满足合并访问, 在此基础上实现了存储数据的合并访问. 本节的主要目的在于实现数据访问时的合并访问.

在算法设计中, 为了减少访存次数, 我们将碰撞与迁移合并, 但两个步骤合并之后, 必须利用 if 语句来谨慎处理边界点的迁移, 否则将直接影响结果的正确性. 另一方面, if 语句又可能引起一个 warp 内的线程跳转到不同的分支, 这将严重影响指令吞吐量. 一旦发生分支, 那么不同的执行路径就必须被串行执行, 这给程序性能带来很大影响.

为了避免使用 if 语句, 我们在网格点四周增加了一层虚拟节点, 这样可以简便地实现碰撞与迁移步骤的合并处理. 需要指出的是, 这样处理同时也带来另外一个问题, 由于在 x 方向增加了网格节点, 对数据的访问不再满足合并访问. 为了克服这一不足, 在 x 方向, 每一行的起始位置由 1 个虚拟格点增加到 16 个虚拟格点, 同时在每行末端补齐, 使每行的格点数是 16 的倍数.

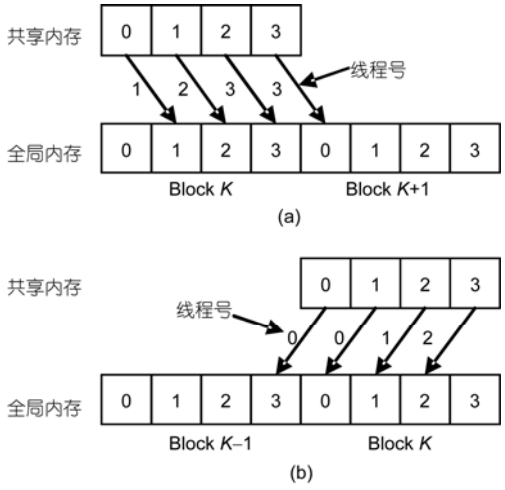

图 4 算法 S3 的数据存储方案

(a) 向右迁移; (b) 向左迁移

(ⅱ) Grid 和 Block 的维度设计. 合适的 Grid 与 Block 尺寸, 将能够最大程度的发挥 GPU 的执行效率. 在程序设计时, 我们将优先考虑 block 的尺寸, 而 grid 的尺寸一般来说越大越好.

在 Tesla 架构下 GPU 的每个 SM 中, 至少要有 6 个 active warp 才能有效地隐藏流水线延迟. 此外, 如果所有的 active warp 都来自同一 block, 当这个 block 中的线程进行存储器访问或者同步时, 执行单元就会闲置. 基于以上原因, 最好让每个 SM 上拥有至少 2 个 active block[17].

对于 S1 来说, 经过计算, 只要每 block 不超过 128 个线程, 即可满足以上要求. 在此, 我们选取了 (16, 2, 1), (16, 4, 1), (32, 2, 1), (32, 4, 1), (64, 2, 1)五种维度设计, 对于 1024×1024 的网格规模, 运算 10000 步, 各自所消耗时间如表 1 所示.

从表 1 中可以看出, (32, 2, 1)的维度设计用时最短, 因此对算法 S1 来讲, 它是最合适的 block 尺寸.

算法 S2 与 S3 仅在存储数据时存在差异, 引入共享内存的目的均是为了实现合并访问, 使用的共享内存、寄存器数量基本一致. 因此, 所适用的 block 尺寸必定相同.

由于执行碰撞迁移的 kernel 运行时间占总运行时间的绝大部分, 在二维情形下, 碰撞迁移 kernel 占用时间在 90%以上. 因此, 我们考虑的重点是碰撞迁移过程的 kernel.

首先考虑 blockDim.x 的影响. 在 S2,S3 中, 在一个 block 所处理的数据中, 其左右两个端点的数据需要特别处理, 而内部的点满足合并访问. 因此在允许范围内, blockDim.x 越大, 需要特别处理的数据所占的比例越小, 更有利于提高程序性能. 基于上述考虑, blockDim.y, blockDim.z 均设置为 1. 对于我们使用的 Tesla C1060, SM 上可用资源如表 2 所示.

表 1 1024×1024 网格下运行 10000 步时间(s)

Block 维度	(16,2,1)	(16,4,1)	(16,8,1)	(32,2,1)	(32,4,1)	(64,2,1)
时间(s)	12.00	11.86	12.05	11.66	12.55	12.26

表 2 Tesla C1060, SM 上可用资源

Warp 总数上限	Block 总数上限	寄存器数量	Shared memory
32	8	32 bit × 16384	16 KB

为达到以上要求, 需满足:

1) 每个 block 最多使用的 shared memory: 16384 *B*/2 = 8192 *B*.

2) 每个 block 最多使用的寄存器数量: 16384/2 = 8192.

接下来, 我们将通过 NVIDIA 公司在 CUDA toolkit 中提供的 CUDA profiler 来分析每个 block 使用的资源数量. 以 S3 为例子, 表 3 给出了 block 维度分别为(128, 1, 1)和(256, 1, 1)时资源占用情况.

由表 3 可以看出, 当 blockDim.*x* 为 128 时, 每个 SM 可以拥有至少两个 active block, blockDim.*x* 为 256 时不满足, 因而大致推断, block 维度为(128, 1, 1)时程序性能可能为最佳(由于采用的格子规模均为 2 的倍数, 为简便起见, 不考虑 128~256 中的数值). 为了验证上述分析, 我们分别列出了几种维度划分在 1024 × 1024 的网格规模下运行 10000 步所消耗的时间, 具体结果见表 4.

表 4 中的结果进一步证实了我们之前的相关分析. 从表中还可以看到, 在 blockDim.*x*×blockDim.*y* 相等的情况下, blockDim.*x* 越大, 所用时间越短; 在(128, 1, 1)维度下, 程序性能达到最优. 在下文中, 对于算法 S2,S3, 均采用该尺寸.

(ⅲ) 调整共享内存和寄存器使用量. 调整 shared memory 和 register 的使用量, 可以均衡资源, 获得更高的 SM 占用率, 从而提高程序性能. 在算法 S2,S3 中, 使用共享内存的目的是为了简便实现在存储数据时满足合并访问. 但是对于在 *x* 方向不发生偏移的方向来说, 其迁移步骤不影响合并访问, 因而可以用寄存器代替共享内存, 减少共享内存的使用量. 在二维模型中, 由于方向较少, 故效果不明显. 本优化技术将在三维程序中使用.

表 3 两种维度下程序占用资源数量

	Shared memory	寄存器数量
(128, 1, 1)	4648	1792
(256, 1, 1)	9256	3584

表 4 1024 × 1024 网格下运算 10000 步时间(s)

Block 维度	(32, 1, 1)	(64, 1, 1)	(64, 2, 1)	(128, 1, 1)	(128, 2, 1)	(256, 1, 1)
S2	13.74	11.47	11.67	10.64	11.71	11.05
S3	10.53	10.03	9.98	9.76	10.76	10.66

3.3 结果分析

实验的硬件配置: CPU 为 Xeon 5520(共 8 核), 8G 内存, GPU 为 Tesla C1060, 有 1 个 GT200 核心, 共 240 个流处理器及 4 GB 图像存储器. 软件平台: 操作系统为 Centos 5.4×64, 程序编译运行环境为 CUDA3.0 及 gcc4.1. 基于 GPU 的两种模型 D2Q9 及 iD2Q9 得到的结果均与文献[21]结果吻合较好.

(ⅰ) 3 种算法性能的对比分析. 为了进一步分析上述 3 种算法的性能, 我们以二维方腔流为例(雷诺数 *Re* = 400, 格子速度 *c*=10, 网格规模分别为 256×256, 512×512, 1024×1024, 2048×2048, 4096×4096), 给出了 3 种算法以每秒百万格子更新(Million Lattice-site Updates Per Second, MULPS)为单位的数值结果, 具体结果见表 5.

从表 5 中可以发现, 在各个网格尺寸下, S1 最慢. 这是因为在二维情形下, 左右边界处理不满足合并访问, 边界处理所占运行时间较少, 因此将边界处理与碰撞迁移在同一个 kernel 中执行其优势较小, 同时在存储数据时不满足合并访问. 算法 S3 在所有情形下均表现最好, 这是由于其在不增加 kernel 的基础上实现了合并访问, 相比 S2, 减少了部分数据访存且处理更简单.

(ⅱ) D2Q9 和 iD2Q9 模型及加速比. 国内相关研究成果所取得的加速比大多在 50 左右. 中国科学院过程研究所李博等人的研究成果, 针对于 2048×2048 的网格, 取得了两百多倍的加速比, 但其作为比较的 CPU 程序速度较慢. 此外, 他们的 GPU 程序运行时间远大于我们的相关结果(见表 6).

国内外 LBM 的 CUDA 实现, 几乎都是基于 D2Q9

表 5 不同算法的性能对比 a)

网格	256×256	512×512	1024×1024	2048×2048	4096×4096
S1	669	776	784	760	766
S2	697	936	986	985	953
S3	771	1012	1074	1080	1067

a) 单位: MLUPS

表 6 D2Q9 模型在 CPU 和 GPU 上运行 10000 步时间(s)

网格	512×512	1024×1024	2048×2048	4096×4096
CPU (s)	370.30	1487.06	5973.18	26744.18
GPU (s)	2.59	9.76	38.84	157.24
加速比	142.95	152.31	153.80	170.09

模型. 在此基础上, 我们实现了文献[19]的 iD2Q9 模型, 取得了比 D2Q9 模型更高的加速比, 具体结果见表 7.

在相同的网格规模下, D2Q9 模型和 iD2Q9 模型在 CPU 上运行时间相差无几, 而在 GPU 上差别则相对较大. 表 8 给出了 D2Q9 模型 iD2Q9 模型运行 10000 步所需时间的比值. 从表中可以看出, 对于 GPU 程序, iD2Q9 模型的效率明显高于 D2Q9 模型, 这主要因为 iD2Q9 模型中 0 方向的数据可以不进行访存. 在 CPU 程序中, 影响程序运行时间的最主要因素是计算部分, 因而差别并不大. 但在 GPU 中, 限制程序性能的瓶颈在于数据访存, 因而, iD2Q9 模型较 D2Q9 模型大约会快 9/8=1.125 倍.

由于加速比涉及到 CPU 的实现, CPU 程序的快慢将直接影响到加速比, 因而, 在某种程度上加速比并不能真实反映 CUDA 程序的快慢. 国外的相关工作大多以每秒百万网格更新率来反映程序的运行速度, 表 9 对比了本文与 Kuznik 等人的数值结果[13].

4 圆柱绕流的模拟

二维方腔流几何结构简单, 且容易选取适当的网格规模, 如 512, 1024 等 2 的次幂, 以适应 GPU 的硬件架构. 因此, 目前国内外基于 CUDA 的 LBM 实现以及对加速比探讨的相关工作几乎均以二维方腔流为例. 然而, 在实际研究中, 许多问题的边界条件较为复杂, 且难以选取“恰好合适”的网格规模. 针对这一情况, 我们模拟了边界较为复杂的圆柱绕流, 其流场结构如图 5 所示: 圆柱管道宽 H=0.41 m, 圆柱直径 D = 0.1 m, 入口边界条件 $U(0, y, t) = 4U_m y(H - y)/H^2$, $V = 0$, $U_m = 1.5$ m/s, 雷诺数定义为 $Re = \bar{U}D/\nu$, 其中 $\bar{U} = 2U(0, H/2, t)/3$.

对于圆柱壁面, 我们使用反弹格式来处理, 其余边界均使用非平衡态外推格式处理. 由于在运用反弹格式的边界处理方法时, 不可避免地要大范围内使用 if 语句, 这将会降低程序性能, 为此, 在碰撞迁移时, 我们不进行反弹处理, 增加一个新的 kernel 专门处理圆柱周围需要反弹的数据.

另外, 我们在模拟过程中选用 iD2Q9 模型以及 S3 算法, 分别在 10 种网格规模下运算 10000 步, 其运行时间及格子更新率如表 10 所示.

由表 10 可见, 当格子规模大于 3281×657 时, 其格子更新率超过二维方腔流的结果, 这可能有两方面原因: 一方面圆柱较小, 圆柱的复杂边界对程序性能影响有限; 另一方面左右边界数据的存取不满足合并访问, 这使得左右边界处理的速度较慢, 而相同网格规模下, 由于采用长方形流场结构, 圆柱绕流左右边界的数据量相比二维方腔流所占比例较小, 有

表 7 iD2Q9 模型在 CPU 和 GPU 上运行 10000 步时间(s)

网格	512×512	1024×1024	2048×2048	4096×4096
CPU (*s*)	366.47	1460.15	6096.35	25680.13
GPU (*s*)	2.32	8.71	34.72	140.63
加速比	157.97	167.66	175.58	182.61

表 8 两种模型的程序运行时间的比值

网格	512×512	1024×1024	2048×2048	4096×4096
CPU(T_{D2Q9}/T_{iD2Q9})	1.01	1.01	0.98	1.04
GPU(T_{D2Q9}/T_{iD2Q9})	1.116	1.120	1.119	1.118

表 9 基于 CUDA 的 LBM 程序效率对比

网格		512×512	1024×1024	2048×2048	4096×4096
本文	D2Q9	1012	1074	1080	1067
	iD2Q9	1130	1204	1208	1193
Kuznik		935	947	909	915

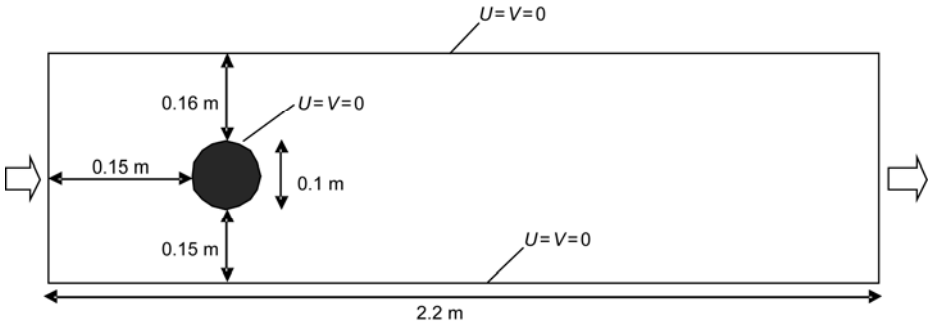

图 5 圆柱绕流流场结构示意图

表 10　圆柱绕流程序不同网格下运行时间及格子更新率

网格	821×165	1641×329	2461×493	3281×657	4101×821	4921×985	5741×1149	6561×1313	7381×1477	8201×1641
时间(s)	1.43	4.73	10.07	17.66	27.47	38.57	54.45	68.15	87.33	111.95
MLUPS	947	1141	1205	1221	1226	1257	1211	1264	1248	1202

助于程序性能提升. 表 11 显示了程序在三种网格下的加速比.

5　三维方腔流的模拟

在现实生活中, 真实流动多为三维流动. 然而, 当维数增加时, 计算量和计算复杂性也随之大大增加, 这对计算方法和计算能力提出了更高的要求. 因此, 提出适应性更广的算法, 开发性能更优的计算程序也就显得颇为重要. 事实表明, 将上述优化的 LB 程序应用到三维问题时, 同样显示出了其突出优势. 不失一般性, 下面以三维顶盖驱动方腔流为例(几何形状如图 6 所示), 我们使用 GPU 程序实现了 iD3Q19 模型中的 S1 算法. 为了验证 GPU 程序的正确性, 我们选取了文献[22]中的参数进行了模拟, 结果与文献[22]吻合很好.

除此以外, 我们还对不同模型(D3Q19, iD3Q19, D3Q15, iD3Q15)、不同算法(S1, S2 S3)在不同网格以及不同结构的 block 下分别进行了实验, 计算时参数取值 Re=200, c= 10, 运行 10000 步, 所得计算时间如表 12~14 所示.

从以上数据中, 我们可以发现以下结论:

(1)对于 S1 算法, 在各种情形下, block 维度为(16, 2, 2)时程序运行时间均少于 block 维度为(32, 2, 2)时的情况. 这可能是由于后者每个 block 占用 shared memory 较多, 前者 Grid 数目较多的缘故.

(2)对于 S2, S3 算法, D3Q15 及 iD3Q15 在网格规模为 $128 \times 128 \times 128$ 时, 最优 block 维度为(128, 1, 1), 其余情形下, 最优 block 维度为(64, 1, 1).

表 11　圆柱绕流程序加速比

网格	821×165	1641×329	2461×493
CPU 运行时间	193.33	761.75	1685.57
GPU 运行时间(s)	1.43	4.73	10.07
加速比	135	161	167

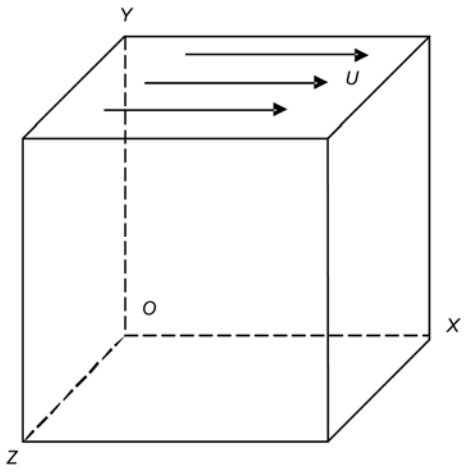

图 6　三维顶盖驱动方腔流的几何形状

表 12　S1 算法运行 10000 步时间(s)

网格维度划分	64×64×64			128×128×128	
	(8,2,2)	(16,2,2)	(32,2,2)	(16,2,2)	(32,2,2)
D3Q19	8.20	7.16	9.86	52.47	70.97
iD3Q19	7.67	6.76	9.29	49.13	66.78
D3Q15	6.48	5.94	6.74	42.06	42.67
iD3Q15	6.05	5.56	6.36	39.80	40.53

表 13　S2 算法运行 10000 步时间(s)

网格	64×64×64			128×128×128			
	(16,2,2)	(32,2,2)	(64,1,1)	(16,2,2)	(32,2,2)	(64,1,1)	(128,1,1)
D3Q19	11.13	11.08	9.59	86.73	82.82	67.82	74.46
iD3Q19	10.25	9.73	8.72	80.93	76.10	61.83	65.93
D3Q15	8.59	7.47	7.07	67.48	54.86	49.79	46.74
iD3Q15	8.02	6.96	6.57	63.66	51.68	46.15	43.40

表 14 S3 算法运行 10000 步时间(s)

网格	64×64×64			128×128×128			
	(16,2,2)	(32,2,2)	(64,1,1)	(16,2,2)	(32,2,2)	(64,1,1)	(128,1,1)
D3Q19	9.47	10.03	8.46	66.71	69.04	57.17	64.82
iD3Q19	8.76	9.47	7.82	61.61	65.80	52.68	61.02
D3Q15	7.44	7.00	6.57	51.69	48.20	43.09	42.61
iD3Q15	6.91	6.50	6.05	48.50	44.86	40.19	39.67

实际上, 经计算, 对于 D3Q19 及 iD3Q19 模型, 在 block 维度为(128, 1, 1)时, 每 SM 内 active block 仅为 1 个, 因此速度较慢, 而(64, 1, 1)的 block 维度更优. 对于 D3Q15 及 iD3Q15 模型, 在 block 维度为(128, 1, 1)时, 仍然满足每 SM 内至少两个 active block 的要求, 正如在上文的分析, 在适合范围内 blockDim.*x* 越大越能体现合并访问的效果, 因此 D3Q15 及 iD3Q15 在网格规模为 128×128×128 时, 最优 block 维度为(128, 1, 1).

(3)在绝大部分情形下, S1 算法的程序运行速度高于 S2,S3 算法的程序, 这和在二维情形下 S3 算法最优的结论相反.

导致这种变化的原因可能有两方面, 一是在三维情形下, 在做边界处理存取边界点数据时, 非合并访问的程度大大加重, S1 算法减少了边界点的访存次数, 因而较有优势; 二是由于网格维度及共享内存的限制, 使得大部分情形下 blockDim.x 最大为 64, 不能充分体现出 S2, S3 算法在存储数据满足合并访问的优势. 同时注意到, D3Q15 及 iD3Q15 在网格规模为 128×128×128 时, S3 算法优于 S1 算法, 这可能正是由于 blockDim.*x*=128 从而合并访问优势较明显的缘故. 而 S2 算法在此情形下仍慢于 S1 算法, 可能是由于其实现合并访问方法较麻烦所致. 这同时也说明了 S3 算法的优越性.

从上文的分析可知, D3Q19, iD3Q19 模型在 block 维度为(128, 1, 1)时程序性能未能提高的原因就在于使用共享内存较多所致. 因此, 针对这一情况, 我们采用 3.2(ⅲ)的优化技术减少共享内存的使用量, 将使用共享内存以实现合并访问的迁移方向降低为 10 个, 优化后在各网格规模及 Block 维度下运行时间如表 15 所示.

从表 15 中可以看到, 对于 D3Q19, iD3Q19 模型, 减少共享内存使用量后 S3 程序的运行时间大部分情形下少于优化之前的程序版本, 其最好结果已优于 S1 程序的最好结果. 而对于 D3Q15, iD2Q15 模型, 在部分 block 维度下, 其速度反而变慢, 最好结果也优于 S1 程序, 但性能提升相对较小, 这可能是由于这两个模型方向较 D3Q19, iD3Q19 模型少, 不需要使用共享内存的迁移方向也较少的原因(D3Q15 有 5 个方向可以不使用共享内存, iD3Q15 仅 4 个方向).

最后, 本文列出了各个模型的最优格子更新率和加速比, 结果如表 16,17 所示.

从表 16 可以看出, 四种模型的格子更新率在 360~550 之间, 这与计算二维方腔流时使用的 D2Q9 及 iD2Q9 模型的格子更新率相差 2~3 倍. 其中一个重要的原因是, 三维模型的方向数是二维模型的 2 倍左右(D2Q9 及 iD2Q9 模型的方向数分别是 9 和 8, 而 D3Q19,iD3Q19,D3Q15,iD3Q15 模型的方向数分别为 19,18,15,14). 方向数的影响主要有两个方面: 第一, 方向数越多, 计算时每个格子点读写的显存数据就

表 15 减少共享内存使用量后 S3 算法运行 10000 步时间(s)

网格	64×64×64			128×128×128			
	(16,2,2)	(32,2,2)	(64,1,1)	(16,2,2)	(32,2,2)	(64,1,1)	(128,1,1)
D3Q19	9.19	8.38	7.64	69.60	58.44	50.65	51.34
iD3Q19	8.68	7.84	7.22	66.20	56.49	47.77	47.08
D3Q15	7.28	7.17	6.40	51.70	49.76	42.00	43.96
iD3Q15	6.81	6.56	5.95	48.09	45.33	39.40	39.23

表 16 各模型最优格子更新率

网格	D3Q15	iD3Q15	D3Q19	iD3Q19
64×64×64	441	471	366	388
128×128×128	499	535	414	445

表 17 各模型运行 10000 步最短时间及加速比

网格大小		64×64×64	128×128×128
D3Q19	CPU	1148.8	8839.6
	GPU	7.16	50.65
	加速比	160	174
iD3Q19	CPU	1108.2	8722.8
	GPU	6.76	47.08
	加速比	164	185
D3Q15	CPU	997.5	7616.8
	GPU	5.94	42
	加速比	168	181
iD3Q15	CPU	941.6	7451.6
	GPU	5.56	39.23
	加速比	169	190

越多, 而显存数据的读写正是 GPU 程序计算速度的一个瓶颈; 第二, 方向数越多, 使用共享内存时每个 block 能容纳的格子点数越少, 由于程序中一个线程处理一个格子点, 所以每个 block 能容纳的线程数越少, 这就导致了 SM 中 SP 的占用率有可能偏低.

另外, 从表 17 可以看出, 采用 GPU 实现的 LB 模型, 能够极大地加快计算速度, 使以前原本需要一百天左右才能完成的计算在一天内就可以完成, 这在计算流体力学中特别是需要高分辨率的网格时(如湍流的数值模拟)显得尤为重要.

6 结论

为解决日趋复杂的大规模科学工程问题, 以计算方法和计算机硬件为基础的高性能计算越来越受到当今世界各国的普遍重视和高度关注, 其发展水平已成为某一国家在现代科学和高技术研究与开发中竞争能力的重要标志. 伴随着计算机软硬件水平的飞速发展, GPU 及其并行计算开发环境 CUDA 的相继出现使得基于 GPU 的并行计算设计逐渐成为高性能计算领域的新宠. 与此同时, 近年来发展起来的格子 Boltzmann 方法具有计算程序实现简单, 天然并行, 易于处理复杂边界等优点, 这些优点使得 LB 方法非常适合利用 GPU 进行大规模模拟计算. 目前, 国内外基于 GPU 的 LB 程序实现已经取得了一定的成绩, 但程序性能有待进一步提高, 许多方面仍缺乏系统探讨. 本文从简单到复杂, 从低维到高维研究基于 CUDA 的格子 Boltzmann 优化计算和数值模拟, 提出了基于 CUDA 程序实现的 S1, S3 算法, 并与国内外研究者广泛使用的 S2 算法进行了比较; 设计了复杂流动的格子 Boltzmann 方法的计算模型, 开发了相应 CUDA 环境下的 LB 并行程序, 模拟了二维方腔流、三维方腔流及边界较复杂的圆柱绕流; 以格子更新率、加速比等为性能评价指标, 综合应用程序设计中的存储器访问优化、资源均衡、细致的任务划分等手段, 详细探讨了 LB 程序的算法优化、GPU 优化等. 特别的, 为减小计算量、方便运用, 我们实现了 iD2Q9 模型、非平衡态外推边界处理方法. 模拟结果显示, 本文获得了 180 以上的加速比, 1200 百万/秒以上的格子更新率, 这有力地证实了 GPU 与 LB 作为大规模并行计算硬件和软件的良好匹配关系, 两者的结合提供了更为理想的加速效果.

致谢 本文的完成过程中柴振华博士、向秀桥博士以及鲁建华博士提出了大量有益的建议, 在此对他们的帮助与支持表示感谢!

参考文献

1 Li W, Wei X, Kaufman A. Implementing lattice Boltzmann Computation on graphics hardware. Vis Comput, 2003, 19: 444–456
2 Chu N, Tai C L. Moxi: Real-time ink dispersion in absorbent paper. ACM Trans Graph, 2005, 24: 504–511
3 Qiu F, Zhao Y, Fan Z, et al. Dispersion simulation and visualization for urban security. IEEE Vis, 2004, 553–560
4 Wei X, Zhao Y, Fan Z, et al. Lattice-based flow field modeling. IEEE Trans Vis Comput Graph, 2004, 10: 719–729
5 Zhu H, Liu X, Liu Y, et al. Simulation of miscible binary mixtures based on lattice Boltzmann method. Comp Anim Virtual Worlds, 2006, 17: 403–410

6 Zhao Y, Wang L, Qiu F, et al. Melting and flowing in multiphase environments. Comput Graph, 2006, 30: 519–528
7 Zhao Y, Han Y, Fan Z, et al. Visual simulation of heat shimmering and mirage. IEEE Trans Vis Comput Graph, 2007, 13: 179–189
8 王国锦, 陈雷霆, 何明耘. 基于 LBM 模型在 GPU 上实时草波动的实现研究. 计算机应用, 2006 , 26: 271–275
9 柳有权, 刘学慧, 吴恩华. 基于 GPU 带有复杂边界的三维实时流体模拟. 软件学报, 2006, 17: 568–576
10 朱红斌, 刘学慧, 柳有权, 等. 基于 Lattice Boltzmann 模型的液液混合流模拟. 计算机学报, 2006, 29: 2071–2079
11 Tölke J, Krafczyk M. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. Int J Comput Fluid D, 2008, 22: 443–456
12 Tölke J. Implementation of a lattice Boltzmann kernel using the Compute Unified Device Architecture developed by Nvidia. Comput Visual Sci, 2008, doi: 10.1007/s00791-008-0120-2
13 Kuznik F, Obrecht C, Rusaouen G, et al. LBM based flow simulation using GPU computing processor. Comput Math Appl, 2010, 59: 2380–2392
14 Obrecht C, Kuznik F, Tourancheau B, et al. A new approach to the lattice Boltzmann method for graphics processing units. Comput Math Appl, 2011, 61: 3628–3638
15 Bernaschi M, Rossi L. Graphics processing unit implementation of lattice Boltzmann models for flowing soft systems. Phys Rev E, 2009, 80: 066707
16 李博, 李曦鹏, 张云, 等. 耦合 Nvidia/AMD 两类 GPU 的格子玻尔兹曼模拟. 科学通报, 2009, 54: 3177–3184
17 张舒, 褚艳丽, 赵开勇, 等. GPU 高性能运算之 CUDA. 北京: 中国水利水电出版社, 2009
18 郭照立, 郑楚光. 格子 Boltzmann 方法的原理及应用. 北京: 科学出版社, 2008
19 He N Z, Wang N C, Shi B C, et al. A unified incompressible lattice BGK model and its application to three-dimensional lid-driven cavity flow. Chin Phys, 2004, 13: 40–46
20 Guo Z L, Zheng C G, Shi B C. An extrapolation method for boundary condition in lattice Boltzmann method. Phys Fluids, 2002, 14: 2007–2010
21 Ghia U, Ghia K N, Shin C Y. High-re solution for incompressible flow using the Navier-Stokes equations and a multigrid method. J Comp Phys, 1982, 48: 387–411
22 Shi B C, Han H F. Parallel lattice BGK simulations of three-dimensional lid-driven cavity flow. In: Proc of WCCM VI in conjunction with APCOM′04, 2004

CUDA based lattice Boltzmann method: Algorithm design and program optimization

HUANG ChangSheng[1], ZHANG WenHuan[2], HOU ZhiMin[1], CHEN JunHui[1], LI MingJing[1], HE NanZhong[1] & SHI BaoChang[1,2]

[1] *School of Mathematics and Statistics, Huazhong University of Science and Technology, Wuhan 430074, China;*
[2] *State Key Laboratory of Coal Combustion, Huazhong University of Science and Technology, Wuhan 430074, China*

Lattice Boltzmann method (LBM) has become a powerful tool in modeling and simulating fluid flows for its fully parallelism, easy implementation, and simple code, and thus, LBM is quite suitable for large-scale computation of fluid flows on graphic processing unit (GPU) due to these advantages. In this paper, we implement the LBM algorithm on GPU using CUDA, and simulate 2D cavity flow, 2D flow around a cylinder and 3D cavity flow. The role of memory access optimization and other optimization technologies in programming and the performance of programs are analyzed in detail. The results show that our algorithm gives satisfactory acceleration, and confirm that LBM is very compatible with GPU for large-scale parallel computation.

lattice Boltzmann method, CUDA, parallel computing, GPU, optimization

doi: 10.1360/972011-1078