科学通报 2009 年 第 54 卷 第 20 期: 3177 ~ 3184
论 文 www.scichina.com csb.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

耦合 Nvidia/AMD 两类 GPU 的格子玻尔兹曼模拟

李博[①②], 李曦鹏[①②], 张云[①], 陈飞国[①], 徐骥[①②], 王小伟[①], 何险峰[①], 王健[①], 葛蔚[①*], 李静海[①]

① 中国科学院过程工程研究所, 多相复杂系统国家重点实验室, 北京 100190;
② 中国科学院研究生院, 北京 100049
* 联系人, E-mail: wge@home.ipe.ac.cn

2009-06-24 收稿, 2009-09-16 接受
国家重大科研装备研制项目(编号: ZDYZ2008-2)、国家自然科学基金(批准号: 20221603, 20490201)和中国科学院知识创新工程项目(编号: KGCX2-YW-124)资助

摘要 利用图形处理单元(graphic processing unit, GPU)进行通用计算近年来得到关注, Nvidia 和 AMD 公司已推出了各自的开发环境 CUDA 和 ASC. 很多计算在 GPU 上的速度远高于目前的 CPU. 格子玻尔兹曼方法(lattice Boltzmann method, LBM)作为一种网格上的粒子方法, 对流动模拟具有良好的内在并行性, 非常适合利用 GPU 进行大规模并行计算. 本文提出了一种耦合 Nvidia 和 AMD 的两类 GPU 完成 LBM 凹槽流模拟的算法, 对于两类 GPU, 在 LBM 的 D2Q9 模型下分别设计了相应的算法和程序, 之后利用消息传递接口(message passing interface, MPI)协议通过多程序多数据流(multi-program multi-data, MPMD)模式使其能够联合计算, 以充分发挥混合 GPU 集群系统的性能. 通过 GPU 和 CPU 程序结果的比较, 证实了 GPU 计算的正确性和所能带来的显著的加速比, 为建设通用大规模 GPU 并行计算平台提供了重要参考.

关键词
GPGPU
格子波尔兹曼
Nvidia
AMD
多程序多数据流
联合计算

近几年, 出于图形处理的需求, 图形处理单元(graphic processing unit, GPU)在硬件设计上采用了超长的流水线和大规模线程并行计算. 这使得 GPU 非常适用于数据相关度较低的大规模并行计算. 利用将需要处理的数据转化为图形处理中的纹理, 进而使用 GPU 来进行计算的方法早在本世纪初就已经出现. 然而, 这种编程模式存在难于掌握、缺乏灵活性以及带宽瓶颈等问题, 很难充分发挥 GPU 的计算能力. 2007 年, Nvidia 公司推出了针对通用计算 GPU(general purpose GPU, GPGPU)的编程平台 CUDA(compute unified device architecture)[1], 而 AMD 公司在其后也发布了 ASC(ATI stream computing)[2]. 这些 GPGPU 开发平台的发布, 使得充分利用 GPU 强大的计算能力(最新的 ATI RV770 单精度浮点峰值已经达到 1.2 TFlops)成为可能.

格子玻尔兹曼方法(lattice Boltzmann method, LBM)是 20 世纪 80 年代中期发展起来的一种流动计算方法[3]. 和传统的建立在连续介质模型上的计算流体力学(computational fluid dynamics, CFD)不同, LBM 以分子动力学和统计力学为理论基础, 从微观尺度上建立离散的速度模型, 在满足质量、动量和能量守恒的条件下, 得到各个网格点的分布函数, 对这些分布函数进行统计, 得到速度、压力等宏观量. 因此, LBM 有算法简单、能处理复杂边界条件、压力能够直接求解、编程容易等优势, 另外其内在的并发性和计算密集, 使得 LBM 十分适合大规模的并行计算. 目前国际上 Tölke[4], Riegel[5], Kaufman[6]等人均利用 CUDA 编程平台实现了各自的 LBM 计算, 能够达到数十倍的加速比.

目前用于通用计算的 GPU 主要有 AMD 和

引用格式: 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
Li B, Li X P, Zhang Y, et al. Lattice Boltzmann simulation on Nvidia and AMD GPUs (in Chinese). Chinese Sci Bull (Chinese Ver), 2009, 54: 3177－3184, doi: 10.1360/972009-1347

Nvidia 两家公司的产品，陆续还会有其他公司的产品投入市场，这些 GPU 的硬件结构和编程环境都有较大的差异，很难用通用的方法编程. 正在设计中的开放计算语言(open computing language, OpenCL)或许能够解决代码通用性的问题，但要获得硬件最好的性能，还是应该针对不同硬件特点使用对应的开发环境和算法. 因此，我们考虑结合各种 GPU 的特点，实现不同 GPU 的异构并行计算. 这种模式将为建设更通用和高效的 GPU 并行计算系统提供有效支撑.

1 算例选择

本文选用串列凹槽流作为试验算例，如图 1 所示，该算例的计算区域下部是一个方腔，上部是一个槽道，顶部存在一个以恒定速度移动的平板. 这实际上是方腔流和 Couette 流的结合. 每个进程负责一个凹槽的计算，多个凹槽串联，并且使用周期边界使凹槽流动形成回路. 对于方腔部分，各个进程之间没有数据交换，全部的数据交换通过槽道部分完成，由此实现一个数据传输量相对较小的算例. 但这个算例在微流动研究中又有相当的普遍性和实用意义，比如作为粗糙壁面上微流动的一个简化模型等.

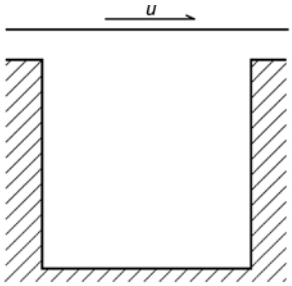

图 1 凹槽流示意图

2 算法实现

我们采用 LBM 的 LBGK 方程按照 D2Q9(D 指维度，Q 指粒子运动方向总数)模型计算整个流场[7]，由此完成了单进程 CPU 程序. 又根据此程序分别发展了基于 Nvidia 和 AMD 的 GPU 并行程序，并利用 MPICH 的 p4pg 模式完成了不同系统的联合计算.

2.1 LBM 的理论基础和 CPU 算法实现

速度分布函数的演化方程，即玻尔兹曼方程[8]为：

$$\frac{\partial f}{\partial t}+\xi\cdot\nabla_x f+\boldsymbol{a}\cdot\nabla_\xi f=\Omega(f)$$

式中：f 为分布函数; $\xi\cdot\nabla_x f$ 为迁移过程项; $\boldsymbol{a}\cdot\nabla_\xi f$ 为外力作用项; $\Omega(f)$部分为碰撞过程.

假设粒子以单一的速率松弛到平衡状态，则得到玻尔兹曼-BGK 方程[8]：

$$\frac{\partial f}{\partial t}+\xi\cdot\nabla f=-\frac{1}{\tau_c}\left[f-f^{(\mathrm{eq})}\right],$$

式中, $f^{(\mathrm{eq})}$为 Maxwell 平衡分布函数.

对玻尔兹曼-BGK 方程进行 Taylor 展开，并进行空间时间离散，得到了 LBGK 方程[9]：

$$f_i(x+c_i\delta_t,t+\delta t)-f_i(x,t)=-\frac{1}{\tau}\left[f_i(x,t)-f_i^{(\mathrm{eq})}(x,t)\right],$$

i 表示粒子运动的第 i 个方向; c_i 为 i 方向上的运动速度; τ 为松弛系数; $f_i^{(\mathrm{eq})}(x,t)$为平衡分布函数.

当平衡分布函数 $f_i^{(\mathrm{eq})}$已知时，才能进一步利用 LBGK 方程计算整个流场. 这里采用常用的 D2Q9 模型[9]，如图 2 所示. 其中 f_i 表示各个方向的分布函数. 该模型中各个方向的平衡分布函数 $f_i^{(\mathrm{eq})}(x,t)$为：

$$f_i^{(\mathrm{eq})}(x)=\varpi_i\rho(x)\left[1+3\frac{\boldsymbol{e}_i\cdot\boldsymbol{u}}{c^2}+\frac{9}{2}\frac{(\boldsymbol{e}_i\cdot\boldsymbol{u})^2}{c^4}-\frac{3}{2}\frac{\boldsymbol{u}^2}{c^2}\right],$$

式中，ρ, $\boldsymbol{u}$ 分别为宏观的密度和流速; c 为基准速度，通常取 1; ϖ_i 为各个方向的权重，当 $i=0$ 时，$\varpi_i=4/9$; 当 $i=1, 2, 3, 4$ 时，$\varpi_i=1/9$; 当 $i=5, 6, 7, 8$ 时，$\varpi_i=1/36$. 密度ρ 和速度 $\boldsymbol{u}$ 分别由以下公式得到.

$$\rho=\sum_{a=0}^{8}f_a,\quad \rho\boldsymbol{u}=\sum_{a=0}^{8}f_a\boldsymbol{e}_a.$$

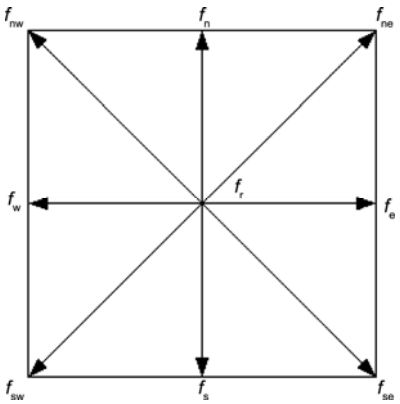

图 2 D2Q9 模型

CPU 上单进程实现 LBM 的算法主要分为碰撞和迁移两部分. 在碰撞函数中涉及多种格点：流体、固体边壁和速度边壁. 对流体格点首先根据分布函数计算密度和各个方向上的速度，并结合两者算出平

衡分布函数，进一步更新各个分布函数；固体边壁上作反弹处理，将该格点的分布函数按相反方向互换，即 f_e 和 f_w，f_n 和 f_s，f_{ne} 和 f_{sw}，f_{nw} 和 f_{se} 互换；而速度边界上则根据设定速度计算得到该格点的平衡分布函数，将这个平衡分布函数作为该点新的分布函数.

迁移时，各个格点将对应方向的分布函数按照图 2 所给箭头方向传给邻近格点. 对于处于边界的格点，由于除了槽道以外其他部分均是边壁，故不需要继续向外迁移；槽道部分则应用周期边界条件使左边和右边连通起来，即最左侧格点的 f_{nw}，f_w 和 f_{sw} 传入最右边格点，最右侧格点的 f_{ne}，f_e 和 f_{se} 传入最左侧格点.

图 3 为 LBM 在 CPU 上单进程实现的算法流程图[10]，首先初始化格点信息(包括分布函数和几何信息)；然后进入迭代过程. 在这个过程中，各个格点独立地更新自己各方向的分布函数，然后迁移分布函数，重复这个迭代过程，直至流场达到稳态或迭代步数达到设定值. 该算法与目前已报道的大部分 LBM 算法[11]是相似的.

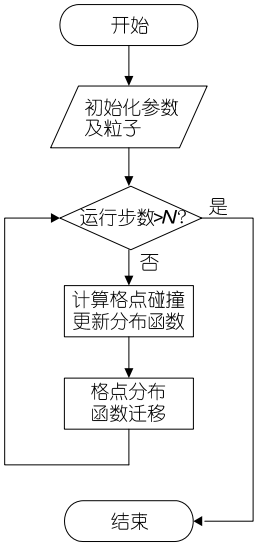

图 3　CPU 单进程算法流程图[10]

2.2　CUDA 算法实现

对于 D2Q9 的 LBM 计算，每个格点的分布函数均含有 9 个分量. 对每个方向的分布函数各建立一个数组，同时考虑内存复制，共创建 18 个一维数组. 另有一个数组作为粒子类型的标识. 多 GPU 并行算法的核心部分借鉴了 Tölke 的单 GPU 算法[4]，与之不同的是内核函数缩减为两个：LBCollProp( )和 LBExchange( )，且依照凹槽流算例的特点对其进行了改动.

LBCollProp( )函数用来处理格点的碰撞和迁移. 不同于 CPU 的算法，此处将二者封装在一个内核函数中，有效地减少了显存读取次数. 整个二维区域先沿 X 方向再沿 Y 方向存入一维数组中. 为提高显存命中率，我们在 3 个方向上按照(*THREAD_NUM*，1，1)将线程组织为线程块(Block)，线程块的栅格(Grid)则为(N_x/*THREAD_NUM*, N_y). 其中 *THREAD_NUM* 表示一个线程块中并发的线程数，N_x 和 N_y 分别是 X 方向和 Y 方向的格点个数. 图 4 是相应的线程块以及栅格划分，其中 *THREAD_NUM* 是 4，N_x 和 N_y 为 8，故线程块是(4，1，1)，线程块的栅格是(2，8)，每一个线程处理一个格点.

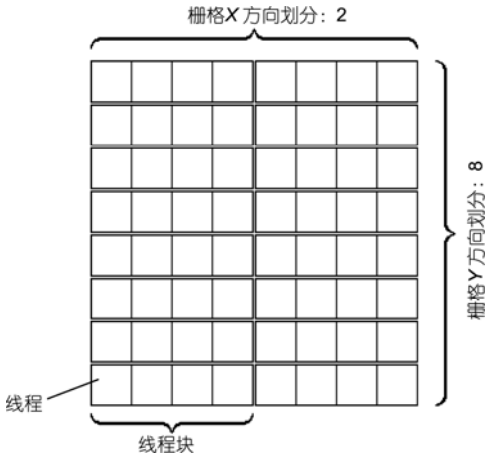

图 4　线程块及栅格的划分

格点尺寸为 8×8，线程块(4, 1, 1)，线程块栅格(2, 8)

在计算和迁移时，使用共享内存来存储需要在 X 方向上迁移的分布函数. 但是这样一来，处于线程块交接处的两个相邻格点就无法通过读取共享内存来交换数据，需要单独的函数 LBExchange( )来处理这些数据，而 Y 方向上迁移的分布函数就没有这个问题.

在 LBExchange( )函数中，需要先后沿 X 正方向和 X 负方向迁移没有到位的分布函数. 由于对每一行内的迁移都需要串行执行(否则将会出现读写冲突)，故将每一行分配给一个线程. 因此线程块仍为(*THREAD_NUM*，1，1)，线程块的栅格改为(N_y/*THREAD_NUM*,1). 同理，考虑到并行进程间的数据迁移，对于 $x=0$ 的最左面一列格点的 3 个分布函数 f_{nw}，f_w 和 f_{sw} 以及 $x=N_x-1$ 的最右面一列格点的 3 个分布函数 f_{ne}，f_e 和 f_{se}，需要另外的两个一维数组分别存储. 如图 5，在迁移时沿 X 正方向，按照 1，2，

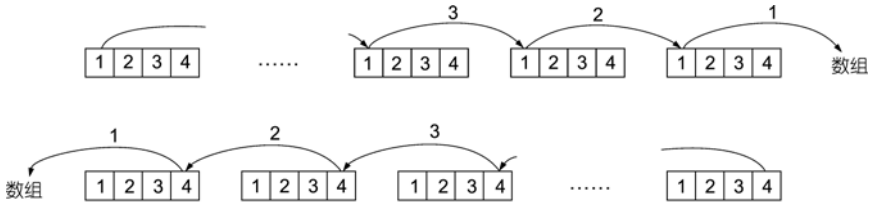

图 5 LBEchange( )函数实现

3…的顺序迁移各个 Block 的 1 号格点的 3 个分布函数 f_{ne}, f_e 和 f_{se}. X 负方向的迁移亦然. 这样, 所有的分布函数均已迁移到了正确的位置.

在 Tölke 的算法中, 还需要处理最左侧格点的 3 个分布函数 f_{ne}, f_e 和 f_{se} 和最右侧格点的 3 个分布函数 f_{nw}, f_w 和 f_{sw}[4]. 而在本算法中, 这一步合并到边界格点的进程间迁移中. 本算法采用 Shift 模式来完成进程间通信. 上文已经提到在 X 边界和 Y 边界均设置了两个一维数组来处理边界迁移, 在进程间通信时只需要传输这 4 个一维数组即可. 对于本算例, 由于只做一维分割, 因此我们省去了 Y 边界的数组和 Y 方向的传递. 整体算法流程图如图 6 所示.

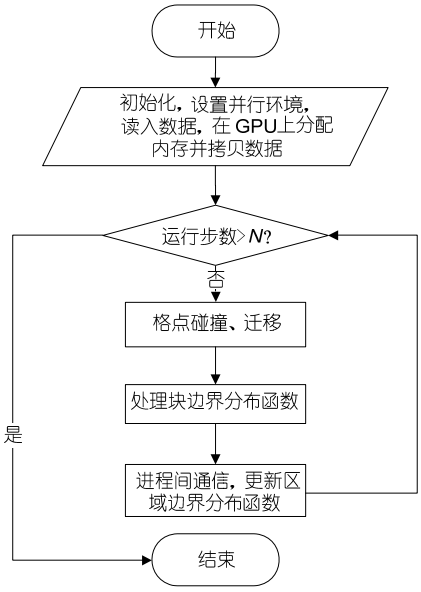

图 6 LBM CUDA 算法流程图

2.3 AMD 算法实现

使用 ASC 的 SDK 可以有两种方法进行编程: CAL[12]和 Brook+. CAL 的编程相对复杂, 而 Brook+ 是基于 C 语言的语法, 更容易掌握, 本文采用 Brook+ 来实现 LBM.

基于 Brook+ 的 LBM 的计算程序主要包含 4 类内核函数, 其中最主要的是 Collision( ), 它用来处理 LBM 的碰撞部分. 在 GPU 的计算中, 9 个分布函数各建立两个流, 共 18 个流. 由于 Brook+ 的内核函数最多只支持 8 个输出流, 因此需要使用两个内核, Collision( )和 Collision_fr( ), 来完成这个工作. 前者完成除本地分布函数 f_r 以外的其他 8 个分布函数的计算, 而后者专门用于处理 f_r. 另外, 这两个内核函数还需要一个整型流 *geoD* 来表示各个格点的类型: 流体(FLUID)、固体(SOLID)和速度边界(SETU). 碰撞内核函数 Collision( )先读取 9 个分布函数, 计算得到当前格点的密度, 然后通过 *geoD* 来判断格点的处理方法, 对于流体先计算各个方向的平衡分布函数, 再根据该格点的平衡分布函数和原有分布函数计算得到这个点新的分布函数; 而固体边界和速度边界的处理与 CPU 算法相同. Collision_fr( )与此类似, 它依然需要读取 9 个分布函数和 *geoD*, 但是它只计算 f_r 这一个分布函数作为输出流. 在 Brook+ 中, 通常的流为只读或只写, 并且由于输入和输出的缓冲区不同, 因此在一个内核中, 输入和输出流不能相同, 因此每个分布函数需要分别建立一个输入流(f_0)和一个输出流(f_1).

其次是 Propagation( ), 它完成 LBM 的迁移部分. 各个格点需要从周围的 8 个格点获得相应的分布函数. 对于并行计算, 需要将不同区域格点划分到不同的进程上, 因此当每个格点更新自己的分布函数并迁移时, 各个划分区域中边界格点的分布函数需要传递给邻近的进程. 在每个时间步计算区域内格点的碰撞和分布函数更新之前, 把邻近进程的边界节点的分布函数传递到本地进程, 因此进程内的格点规模就由原来的 $N_x \times N_y$ 变成了 $(N_x+1)\times(N_y+1)$, 增加了一层虚拟网格. 共需要处理四个边界. 以 f_e 为例, 它需要从左边$(x-1, y)$读取, 但是对于最左边的虚拟网格格点, 无法从更左边获得数据. 因此需要判断, 当需要处理的是最左边的格点时, 则读取当前格点自己的 f_e 作为迁移后的 f_e. 这是由于 Brook+ 中普通流不能支持写入到流的任意位置, 而只能写到流的当

前位置. 这样, 与其说是各个格点将数据传给周围的格点, 不如说是当前格点去周围格点抓取自己需要的数据. 同时考虑过多的判断会降低 GPU 的计算速度, 因此使用三目运算(? :)来取代判断. 例如,

coord.x = (index.x > 0) ? (index.x–1) : index.x;
coord.y = index.y;
fe0 = fe1[coord.y][coord.x].

对于最左边的虚拟网格, 它的作用是接收其右边格点传过来的有用数据, 并打包传递给左边的进程. 然而左边邻近进程并不需要 f_e 这个数据, 因此当前进程的左边虚拟网格中的 f_e 实际是无用的, 最左边的真实网格需要从虚拟网格中获得 f_e, 然而这个数据同样是不真实的, 需要在其后的处理中使用左边进程传递过来的真实数据进行替换. 对于 f_r 这个值, 它并不需要向周围格点传递, 但是同样出于输入输出流必须分离的原则, 使用 Copy( )函数完成当前格点 f_{r1} 到 f_{r0} 的“传递”过程.

利用虚拟网格完成与周围进程进行数据交换需要两个函数：PushtoBuffer( )和 AddtoGrid( ), 前者用于将虚拟网格中对应值拷贝到用于传输的 Buffer 里面, 而后者用于将从周围进程接收到的存入 Buffer 中的数据添加到对应的格点中. 由于 Brook+ 只能将数据写到输出流的当前位置, 而这个位置和执行区域是重叠的, 因此如果只对部分格点进行写入操作, 就需要将内核函数的执行区域设置为与这部分格点相同. 在 AddtoGrid( )中, 设置执行区域为需要添加数据的格点.

与邻近进程的数据交换需要如下几个步骤：

(1) 经过 Propagation( )后, 虚拟网格中已经拥有了需要传递的数据; (2) 利用 PushtoBuffer( )函数, 将这些数据写入一个 Buffer; (3) 利用流处理命令 StreamWrite( )将 Buffer 的内容从显存写入内存; (4) 利用 MPI 协议, 将数据传递给邻近进程; (5) 利用 StreamRead( )将接收到的数据从内存写回显存; (6) 利用 AddtoGrid( )将传入的数据添加到原格点中.

2.4 两类 GPU 的联合算法实现

两类 GPU 由于在硬件上和开发环境上有很大的不同, 因此对于 LBM, 很难有一个统一的算法来实现. 传统上, 这类多进程的并行更多是使用单程序多数据流(SPMD), 而面对两个差异如此大的平台, 这种 SPMD 显得很难适应. 因此, 我们使用多程序多数据流(MPMD)的方式, 即在不同的平台上, 使用不同的程序. 使用 SPMD, 每个进程执行的是相同的命令, 对于每个进程, 其他进程其实是一个白箱, 它所执行的命令即是其他进程执行的命令. 这一模式最大的优点在于编程容易, 只需要编写一套程序. 而使用 MPMD, 每个进程对于其他进程都是一个黑箱, 它所能看到的是一个统一的接口, 各个进程内部可以针对其加速卡的不同特点进行算法优化.

为了实现两类 GPU 的联合计算, 我们通过一个小实验来证明这种可能性. 设 CUDA 程序为 P_1, AMD 程序为 P_2, CPU 程序为 P_3. 对于各不相同的矩阵 A_1, A_2, A_3 和 B_1, B_2, B_3, 令 P_1 计算 $A_1+B_1=C_1$, P_2 计算 $A_2+B_2=C_2$, P_3 计算 $A_3+B_3=C_3$, 然后通过进程间通信把 C_1 传至 P_2 和 P_3, C_2 传至 P_1 和 P_3, C_3 传至 P_1 和 P_2, 令 3 套程序分别计算 $C_1+C_2+C_3=D$. 结果 3 套程序算得的结果 D 一致, 说明不同平台开发的程序确实可以联合计算. 该测试中, P_1, P_2 和 P_3 相当于 3 个黑箱, 它们统一的通信接口是矩阵 C_n, 这种“黑箱+接口”的模式是联合计算的核心思想.

在凹槽流算例中, 我们通过前述算法保证了边界上每个方向只需要传递 3 种数据——向左边是 f_{nw}, f_w 和 f_{sw}, 向右边是 f_{ne}, f_e 和 f_{se}——就可实现 Nvidia 和 AMD 系统间并行计算同一算例. 传递时需要开辟两个内存区域用来发送和接收数据, 每个内存区域的大小是 $3\times h$, 其中 h 为槽道高度, 在这个区域中 3 种数据依次排列, 如图 7 所示. 传递的顺序是先左边后右边. 前面已经提到过, 完成迁移步骤后, 最右侧格点的 f_{nw}, f_w 和 f_{sw} 和最左侧格点的 f_{ne}, f_e 和 f_{se} 都不是真实的数据, 直到这一步, 才由相邻进程传过来的数据所替代.

发起算例时, 利用 MPICH 提供的 p4pg 模式, 编写如下的 pgfile 文件:

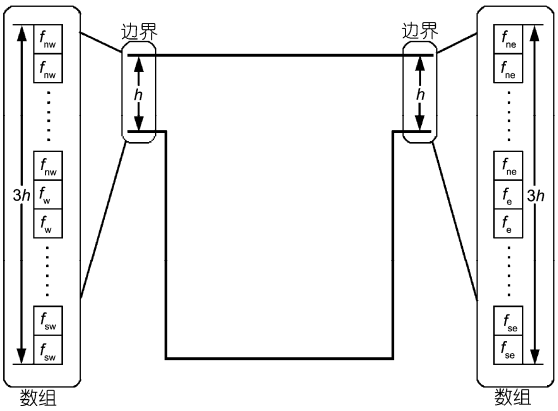

图 7 内存排布

```
hostname1   0   programname1
hostname2   1   programname2
  ……
hostnameN   1   programnameN
```

其中, hostname1~hostname*N* 为进程所在节点名, programname1~programname*N* 为相应的进程名. 然后在 hostname1 节点上输入以下命令

```
mpirun -p4pg pgfile programname1
```

以此来发送不同系统联用的程序.

3 结果分析

作为中国科学院过程工程研究所 Mole-8.7 系统定型前系列测试的一部分, 我们严格检验了上述模型与算法的可行性与正确性. Mole-8.7 系统是该所于 2009 年 4 月建成的国内首套单精度浮点峰值速度达到 1Pflops 的超级计算系统[13], 其中采用了 Nvidia GTX295, GTX280, Tesla C1060 和 AMD HD4870x2 共 4 种 GPU 加速卡. 整个测试分为单 CPU、单 GPU、单系统多 GPU、全系统几大类. 由于存在多核 CPU 和封装多个 GPU 芯片的 GPU 卡, 为避免混淆, 我们统一用"单核"来表示实际对应一个进程的计算硬件. 每个进程(对应单核)的计算规模均为 2048×2048 个格点, 总规模由总进程数确定. 下面分别就此算法的效率和正确性介绍测试结果.

3.1 加速性能

此次测试为了提高 CPU 程序和 GPU 程序的可比性, 在 Xeon E5430 上依照 Intel 处理器的特性做了优化, 特别是对程序进行了向量化处理. 编译采用 10.1.015 版 icc 并采用了-fast 优化选项. 此外, 所有的程序中均采用了 Kahan 算法[14]来提高单精度浮点数在累加时的精度. 对于 CUDA 程序, 采用 1.1 版的 Toolkit 和 SDK, 显卡驱动版本号 180.22; 对于 Brook+ 的程序, 采用 1.4 的 SDK, 显卡驱动版本号 9.4.

表 1 中计算核心指的是碰撞和迁移模块, 而通信模块占用的时间包括了进程之间数据传输以及等待. 表中浮点性能的计算方法为: 格点数×浮点操作数×步数/耗时. Nvidia 和 AMD 的 GPU 联合计算取的是 Nvidia 节点上的数据. 由表 1 中我们可以看出, 单从计算性能上来看, 无论是 Nvidia 还是 AMD 的 GPU, 单核计算能力均能达到 CPU 单核的上百倍, 即使在整个系统的联测时, GPU 单核的整体性能也有 CPU 的 70 倍以上, 性能提升十分可观. GPU 单核的整体联测性能相较单种 GPU 性能的差距部分来自通信开销, 但主要是 GPU 进程间的等待造成的. 虽然 AMD 的 GPU 单核理论峰值高于 Nvidia 的 GPU 单核, 但是由于硬件构造不同, AMD 的 GPU 在读取线性内存时无法做到 Nvidia 那么高的全局内存命中率, 使得大部分的时间花在显存读取上, 实际速度较低, 从而出现了 Nvidia 的 GPU 等待 AMD 的 GPU 的情况, 降低了整体速度. 下一步如能通过负载平衡来尽量减少等待时间, 本算例联算时总体性能应接近 310Gflops/GPU, 即达到约 31%的实际使用效率.

3.2 正确性

上述算例已经在 GPU 系统上取得了较高性能, 但还需要考察计算结果的正确性. 将 CPU 程序算得的结果和两种 GPU 程序算得的结果做比较, 得到速度矢量图(图 8). 在图 8 中, 上部槽道中的流体由于受

表 1 串列凹槽流程序实测报告

测试平台	Intel Xeon E5430 2.66GHz (使用单核)	Nvidia GTX295 (使用单核)	AMD HD4870x2 (使用单核)	232× Nvidia GTX295	60×AMD HD4870x2	232×NvidiaGTX295 + 137×AMD HD4870x2 + 153×Nvidia GTX280 + 62×Tesla C1060
核数	1	1	1	464	120	953
单精度理论峰值(总 Gflops)	21.28	894.24	1200.00	414927.36	144000.00	945281.28
计算核心耗时(s/10^4 步)	28393.36	123.54	209.36	123.73	210.86	131.25
通信耗时(s/10^4 步)	0.00	0.00	0.00	11.48	18.65	253.15
总耗时(s/10^4 步)	28393.36	123.54	209.36	135.21	229.51	384.40
单进程单步 flop	1136	1136	1136	1136	1136	1136
计算核心性能(Gflops/单核)	1.68	385.68	227.59	385.09	225.97	363.03
总体性能(Gflops/单核)	1.68	385.68	227.59	352.39	207.60	123.95
计算部分加速比	1.00	229.83	135.62	229.48	134.66	216.33
总加速比	1.00	229.83	135.62	209.99	123.71	73.86
效率	7.89%	43.13%	18.97%	39.41%	17.30%	12.50%

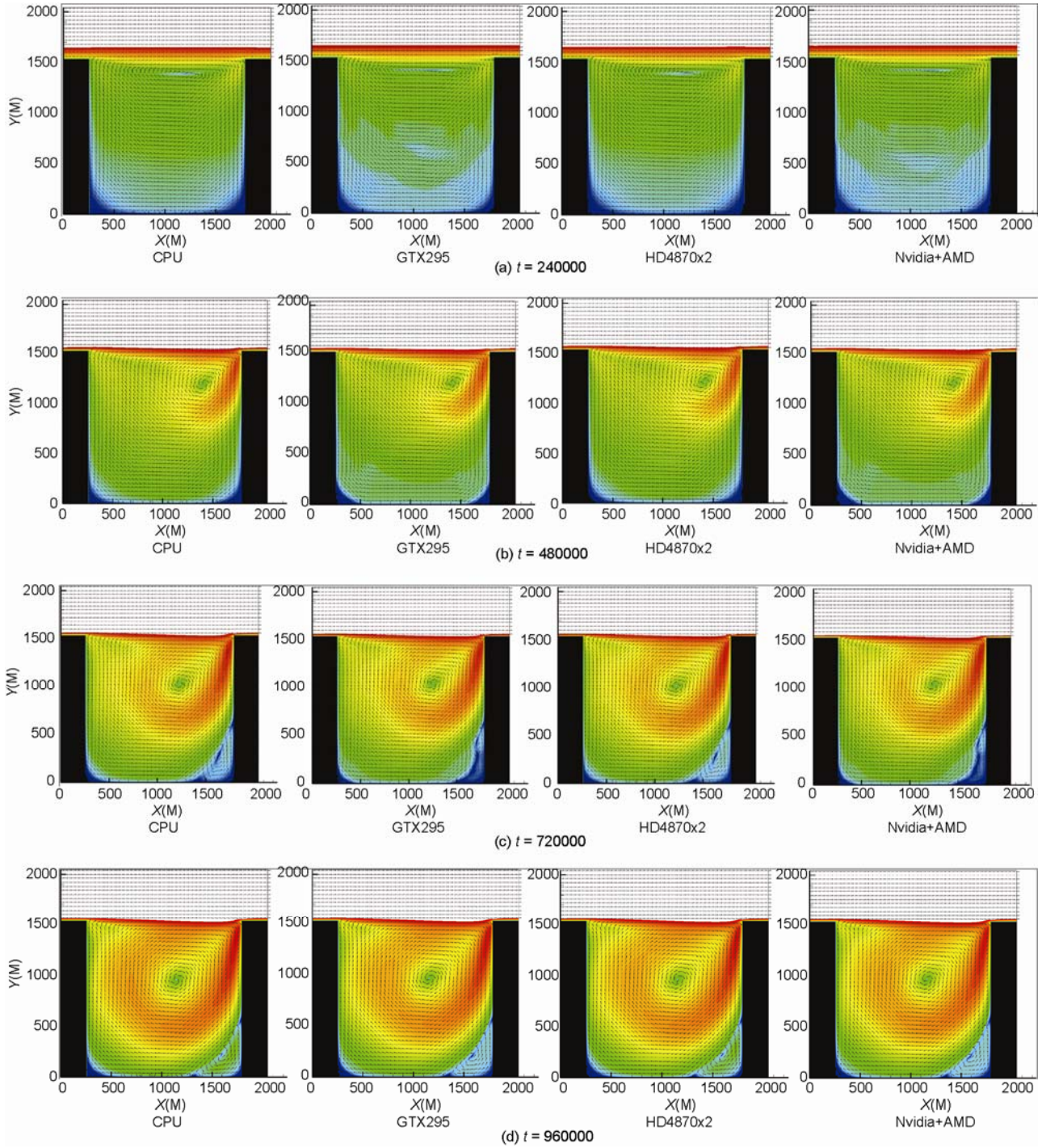

图 8　算例实测结果矢量图

到上板拉动的影响，速度远大于下部方腔中流体速度．下部方腔中流体受到槽道流体作用和固壁限制，在中央形成一个涡流．单从图上来看，几种结果的差别极小．在利用 Origin 8 对计算结果进行分析后得出，各种 GPU 算得的结果和 CPU 相比的最大误差不超过 7×10^{-4}，大部分误差不超过 3×10^{-4}，应该主要是由舍入误差造成的，说明了该计算模式的正确性．

4　结论

利用 GPU 强大的密集计算能力和很高的内存访问带宽，可以大大提高 LBM 的计算速度．同时，采

用 MPMD，使得 Nvidia 和 AMD 的 GPU 可以联用，并能够针对各个平台的特点选用适宜的算法，充分发挥混合 GPU 集群系统的计算能力. 但实现没有充分考虑负载平衡问题，这将是今后工作的重点.

致谢 感谢 Intel 公司陈健在 CPU 程序调优上的帮助.

参考文献

1. Nvidia. Nvidia CUDA Compute Unified Device Architecture Programming Guide. 2007
2. AMD. AMD Stream Computing-User Guide v1.1. 2008
3. Chen S, Doolen G D. Lattice Boltzmann method for fluid flows. Annu Rev Fluid Mech, 1998, 30: 329—364[doi]
4. Tölke J. Implementation of a lattice Boltzmann kernel using the Compute Unified Device Architecture developed by Nvidia. Comput Visual Sci, 2008, [doi]
5. Riegel E, Indinger T, Adams N A. Implementation of a Lattice-Boltzmann method for numerical fluid mechanics using the Nvidia CUDA technology. Comp Sci-Res Dev, 2009, 23
6. Kaufman A, Fan Z, Petkov K. Implementing the lattice Boltzmann model on commodity graphics hardware. J Stat Mech-Theor Exp, 2009, 6: P06016[doi]
7. Succi S. The Lattice-Boltzmann Equation for Fluid Dynamics and Beyond. Oxford: Clarendon Press, 2001
8. Harris S. An Introduction to the Theory of the Boltzmann Equation. New York: Dover Publications, 2004
9. Qian Y H, D'Humieres D, Lallemand P. Lattice BGK models for Navier-Stokes equation. Europhys Lett, 1992, 17: 479—484[doi]
10. 多相复杂系统国家重点实验室多尺度离散模拟项目组. 基于 GPU 的多尺度离散模拟并行计算. 北京: 科学出版社, 2009
11. Sukop M C, Thorne J D T. Lattice Boltzmann Modeling: An Introduction for Geoscientists and Engineers. Berlin: Springer, 2006
12. AMD. Compute abstraction layer (CAL) technology. Intermediate Language(IL) -Reference Manual v2.0, 2008
13. Chen F, Ge W, Guo L, et al. Multi-scale HPC system for multi-scale discrete simulation—Development and application of a supercomputer with one petaflops peak performance in single precision. China Particuology, 2009, 7: 332—335
14. Kahan W. Further remarks on reducing truncation errors. Commun ACM, 1965, 8: 40[doi]

Lattice Boltzmann simulation on Nvidia and AMD GPUs

LI Bo[1,2], LI XiPeng[1,2], ZHANG Yun[1], CHEN FeiGuo[1], XU Ji[1,2], WANG XiaoWei[1], HE XianFeng[1], WANG Jian[1], GE Wei[1] & LI JingHai[1]

[1] State Key Laboratory of Multiphase Complex Systems, Institute of Process Engineering, Chinese Academy of Sciences, Beijing 100190, China;
[2] Graduate University of Chinese Academy of Sciences, Beijing 100049, China

General purpose computing on graphic processing units (GPUs) has received great attention recently. Both Nvidia and AMD have announced their own SDK, CUDA and ASC, respectively. Compared with CPUs, many applications can achieve high speed-up on GPUs. As a mesh-based particle method for flow simulation, lattice Boltzmann method (LBM) has perfect intrinsic parallelism that is very suitable for large-scale parallel computing. In this article, we put forward an algorithm for LB simulation of flow in grooved channel using the D2Q9 model, running on both Nvidia and AMD GPUs in the multi-program multi-data (MPMD) mode through message passing interface (MPI). The capability of hybrid GPU clusters can thus be fully utilized. The correctness and performance of the computation is analyzed through comparison with corresponding CPU implementation and significant speed-up of the GPU implementation has been demonstrated. The results provide valuable references to the establishment of GPU-based high performance computing (HPC) systems.

GPGPU, lattice Boltzmann, Nvidia, AMD, multi-program, multi-data, coupling computing

doi: 10.1360/972009-1347