# 耗散粒子动力学 GPU并行计算研究*

林晨森[1)] 陈硕[1)†] 李启良[2)] 杨志刚[2)]

1) (同济大学航空航天与力学学院, 上海 200092)

2) (同济大学上海地面交通工具风洞中心, 上海 201804)

( 2013 年 12 月 10 日收到; 2014 年 1 月 13 日收到修改稿 )

研究了耗散粒子动力学基于计算统一设备架构的图形处理器 (GPU) 并行计算的实施. 对其中涉及的算法映射模型、Cell-List 法数组的并行化更新、随机数生成、存储器访问优化、负载平衡等进行了详细的讨论. 进一步模拟了 Poiseuille 流动和突扩突缩流动, 从而验证了 GPU 计算结果的正确性. 计算结果表明, 相对于基于中央处理器的串行计算, 在耗散粒子动力学中实施 GPU 并行计算可以获得约 20 倍的加速比.

**关键词**: 耗散粒子动力学, 计算统一设备架构, 图形处理器, 并行计算

**PACS:** 47.61.–k, 47.11.–j, 87.15.N–, 47.10.–g **DOI:** 10.7498/aps.63.104702

## 1 引 言

介观尺度下流体的动力学特征对于生物工程、环境工程、微流控器件设计等领域具有重要的研究价值[1−9]. 耗散粒子动力学 (DPD) 是一种介观尺度的数值模拟技术. DPD 方法结合了分子动力学、格子气自动机和格子-玻尔兹曼方法的优点, 可以模拟较大的时空尺度并具有无网格的特点. DPD 方法已被成功用于模拟流体流变特性、多相流体流动、红细胞变形运动、DNA 分子悬浮物流动等. 尽管如此, DPD 是基于粒子的数值模拟方法, 在对介观尺度数值模拟的研究中, 仍然面临计算耗费大的问题. 为了能够将 DPD 应用到较大的复杂系统 (如毛细血管网络等), 十分有必要在 DPD 中采用高性能计算技术.

图形处理器 (GPU) 是计算机的图形处理单元, 具有成百上千的计算核心, 适用于密集型、高度并行化的科学计算. 基于 GPU 的计算统一设备架构 (CUDA) 降低了 GPU 通用计算开发难度. 李清都等[10] 利用 CUDA 加速了连续时间系统二维不稳定流形的计算, 并进一步通过 CUDA 对简单行走模型的吸引区域进行中央处理器 (CPU) +GPU 的并行计算研究[11]. 汪先超等[12] 利用 GPU 并行计算大幅度减少了计算机断层成像 (CT) 系统数据采集的时间, 可减少辐射剂量, 提高 CT 系统的寿命. 针对流体模拟粒子方法的 GPU 并行计算研究, 目前较多的工作是围绕格子-玻尔兹曼方法展开[13−18]. 郑彦奎等[14] 实现了格子-玻尔兹曼方法的 GPU 并行计算, 并模拟了方腔顶盖驱动流动. 张丹丹等[15] 对格子-玻尔兹曼方法实现了 CUDA, MPI+CUDA, MPI+OpenMP+CUDA 多级并行算法.

目前, DPD 方法 GPU 并行计算的研究处于起步阶段. Wu 等[19] 提出了 DPD 中适合 GPU 并行计算的 Cell-List 更新算法; Wang 等[20] 对多 GPU 并行计算中, DPD 的区间划分、GPU 间通信开销等进行了分析. 本文基于 CUDA 平台, 对 DPD 方法进行了 GPU 并行计算研究. 并就其中的算法映射模型、Cell-List 的并行化更新、随机数生成、存储器访问优化、负载平衡等进行了详细的讨论. 通

* 中央高等学校基本科研基金 (批准号: 125065)、国家自然科学基金 (批准号: 51276130, 10872152) 和教育部高等学校博士学科点专项科研基金 (批准号: 20120072110037) 资助的课题.

† 通讯作者. E-mail: schen_tju@tongji.edu.cn

© 2014 中国物理学会 Chinese Physical Society 

过模拟Poiseuille流动和突扩突缩流动验证了DPD方法GPU并行计算结果的正确性和有效性, 这为DPD进一步实施大规模流体模拟提供了可能.

## 2 DPD

在DPD系统中, 一个粒子代表许多分子或原子的集合. 所有相互作用的粒子都遵循牛顿运动方程. 每个粒子所受合力由保守力、耗散力、随机力三部分组成. 这些力只有在截断半径$r_c$内才发生相互作用. 其中, 保守力$\boldsymbol{F}^{\mathrm{C}}$是一种软作用力, 与粒子之间的距离成反比, 其作用形式为

$$F_{ij}^{\mathrm{C}} = \alpha_{ij}(1 - r_{ij})\boldsymbol{e}_{ij}, \tag{1}$$

其中, $\alpha_{ij}$为排斥力系数, 表示粒子$i$与粒子$j$之间的最大作用值; $r_{ij}$为粒子$i$与粒子$j$之间的距离, $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$; $\boldsymbol{e}_{ij}$为$\boldsymbol{r}_{ij}$的单位矢量, $\boldsymbol{e}_{ij} = \boldsymbol{r}_{ij}/|\boldsymbol{r}_{ij}|$.

耗散力$\boldsymbol{F}^{\mathrm{D}}$作用形式为

$$\boldsymbol{F}_{ij}^{\mathrm{D}} = -\gamma w^{\mathrm{D}}(r_{ij})(\boldsymbol{e}_{ij} \cdot \boldsymbol{v}_{ij})\boldsymbol{e}_{ij}, \tag{2}$$

其中, $w^{\mathrm{D}}$为依赖于粒子间相对距离的权重函数; $\boldsymbol{v}_{ij}$为粒子之间的相对速度, $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$; $\gamma$为耗散系数. (2)式中负号表明耗散力的作用方向总是与相对速度方向相反.

如果系统中只有保守力和耗散力, 那么系统的能量将不能维持平衡, 因此引入第三种力 (随机力)$\boldsymbol{F}^{\mathrm{R}}$以补充系统能量, 其作用形式为

$$\boldsymbol{F}_{ij}^{\mathrm{R}} = \sigma w^{\mathrm{R}}(r_{ij})\theta_{ij}\boldsymbol{e}_{ij}, \tag{3}$$

其中, $w^{\mathrm{R}}(r_{ij})$为依赖于粒子间相对距离的权重函数; $\sigma$为随机力系数; $\theta_{ij}$为随机变量, 其平均值为零且方差为$\Delta t^{-1}$, $\Delta t$为时间步长. 随机力的参数必须与耗散力的参数相匹配, 否则将导致系统能量失衡.

$$w^{\mathrm{D}}(r_{ij}) = [w^{\mathrm{R}}(r_{ij})]^2, \tag{4}$$

$$\sigma^2 = 2\gamma k_{\mathrm{B}}T, \tag{5}$$

其中, $k_{\mathrm{B}}$为玻尔兹曼常数, $T$为平衡态温度. 上述三种作用力的方向均在两个粒子的连线方向上. 目前DPD采用已改进的velocity-Verlet算法更新每个粒子在新时刻的位置、速度和受力情况.

## 3 CUDA

在CUDA平台上GPU相当于CPU的协处理器, 可处理大量相同的指令. 在其架构下, 一个程序被分成host端和device端两部分. host端在CPU上执行, device端在GPU上执行. Device端的代码也叫kernel. 虽然发送到不同线程上的代码相同, 但由于每个线程有自己的编号, 利用这个编号就可以实现对不同数据的操作, 这个体系结构也称为单指令多线程架构.

## 4 DPD中GPU并行计算的实施

### 4.1 粒子间作用力的GPU并行化计算

在DPD中, 恰当选取粒子间相互作用力的计算方法将在很大程度上影响整个模拟所花费的时间. 元胞分割法中的Cell-List方法仅计算截断半径$r_c$内粒子的相互作用力, 避免了求解全场粒子间的相互作用, 可以大幅度节约计算工作量, 是DPD常用的方法之一[21].

以图1为例, 当利用基于CPU的Cell-List方法计算粒子间相互作用力时, 所形成的Cell-List数组列于表1. 计算编号为1的粒子的受力时需要周围元胞内的粒子信息, 具体读取过程如图2所示, 若遇数组元素为空(即其值为零), 则停止计算. 这种方法利用最小的数组空间存储所有粒子的相互作用信息, 并且保证不会随着单个元胞内粒子的增多而溢出.

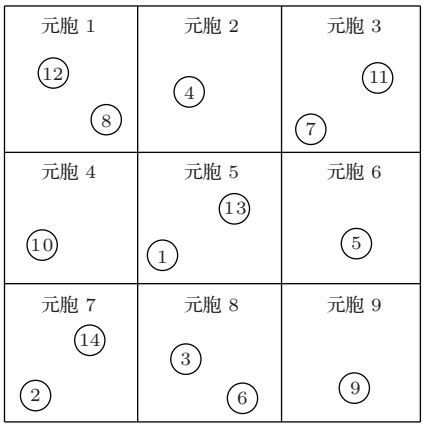


图1 粒子和元胞位置的示意图

上述算法在CPU计算中十分有效, 但在实施GPU并行计算时存在缺陷. 由于创建Cell-List是

一个逻辑串行的过程, 所以无法在GPU上直接并行加速. 而如果每次将粒子位置信息拷贝回host端由CPU 处理, 然后再拷贝回device端进行后续计算, 将严重影响计算效率. 另一方面, 如果在GPU上单独由一个线程进行Cell-List的更新, 将空置GPU的大量计算资源. 因此必须改变Cell-List的构造方法使其适应并行化计算.

本文创建了一个固定大小的二维数组, 不同元胞对应数组中不同的列, 而同一个元胞下的数组元素则包含了处于这个元胞中的粒子编号. 更新完成后的二维数组列于表2. 此种方法能够有效地实施Cell-List方法GPU并行计算, 但是需要比CPU计算中Cell-List更大的数据空间, 并且需要预先估计整个计算过程中单个元胞内可能出现粒子的最大个数, 从而确定出每个元胞下数据元素的最大容量.

表1 Cell-List数组存储表

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 数组A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 7 | 8 | 1 | 2 |
| 数组B | 元胞1 | | 元胞2 | | 元胞3 | | 元胞4 | 元胞5 | | 元胞6 | 元胞7 | | 元胞8 | 元胞9 |
| | 12 | | 4 | | 11 | | 10 | 13 | | 5 | 14 | | 6 | 9 |

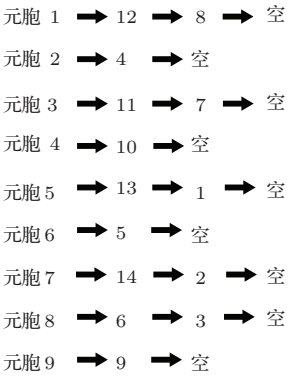


图2 Cell-List读取方式

在每个新的时间步下, Cell-List数组需要更新. 每个线程根据粒子的位置信息计算出一个粒子所在的元胞编号, 然后将该粒子编号写入Cell-List二维数组中. 不同的线程对应不同的粒子, 对于同一个元胞中的不同粒子会出现线程竞争. 由于几个线程可能会同时完成计算, 进而导致数据互相覆盖. 本文采用原子操作使得线程间的读写操作互斥, 每个线程可以依次将粒子编号写入Cell-List数组, 进而完成Cell-List的GPU并行化更新.

采用原子操作的优化算法后, Cell-List数组的更新速度达到约$2 \times 10^9$粒子/s. 在65536个粒子的计算规模下每次更新只需要31 μs, 在524288个粒子的计算规模下也只需278 μs完成更新, 均只占每步总计算时间的0.5%以下. 所耗时间甚至少于将Cell-List全部置零所花费的时间, 因此此算法非常高效.

表2 并行模式下Cell-List二维数组的存储形式

| 元胞1 | 元胞2 | 元胞3 | 元胞4 | 元胞5 | 元胞6 | 元胞7 | 元胞8 | 元胞9 |
|---|---|---|---|---|---|---|---|---|
| 8 | 4 | 7 | 10 | 1 | 5 | 2 | 3 | 9 |
| 12 | 0 | 11 | 0 | 13 | 0 | 14 | 6 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## 4.2 线程映射关系

针对DPD系统的GPU并行计算, 可以采用以下两种方法: 一种是将线程映射到元胞, 一个线程负责计算一个元胞内所有粒子的受力, 所需要的线程数是元胞总数; 另外一种方法是将线程映射到粒子, 一个线程仅负责一个粒子的受力, 所需要的线程数是粒子总数. 如果不同元胞内粒子数目有较大波动, 那么采用第一种算法将导致warp内线程的互相等待, 而第二种算法可缓解这种问题. 经过测试, 线程映射粒子比线程映射元胞的算法要快.

## 4.3 随机数生成的并行化

DPD模拟中, 随机力的计算等需要大量的随机数. 在串行计算时, 可采用时间作为种子, 由编译器内建函数产生伪随机数序列. 在GPU并行计算

中, 如果由host端产生随机数序列, 然后拷贝到显存上, 将耗费大量时间, 从而导致随机数的获得成为DPD方法实施GPU并行计算的瓶颈. 文献[22, 23]提出了不同线程根据不同的种子分别产生随机数序列的粒子系统随机数生成方法. Nandapalan等[24]分析了不同算法在GPU上生成随机数的质量, 并测试了GPU生成随机数的速度. 下面讨论两种均布随机数的生成方法, 非均布的随机数序列可以通过变换获得.

方法一对应的随机数发生器形式如下:

$$x_n = (ax_{n-1} + c)\mathrm{mod}m, \tag{6}$$

$$u_n = x_n/m, \tag{7}$$

其中, $m$为算法周期, mod运算表示将两个数相除并只返回余数. $m$, $a$, $c$分别取为$2^{32}$, 69069, 1234567[25]. 此随机数输出范围为[0, 1). 合并两个随机数发生器可以有更大的周期,

$$v_n = (u_{1n} + u_{2n})\mathrm{mod}1. \tag{8}$$

在串行程序中, 所有粒子共用一个随机数序列, 根据CPU计算的先后顺序依次取用. 在GPU并行计算中, 不同线程采用不同随机数序列. 不同线程间的随机数序列种子不同. 因为随机数算法相同, 所以要求不同线程采用相同随机数序列上的不同片段, 如图3所示. 这里$r_{1,1}$代表随机数, 第一个下标代表元胞编号, 第二个下标代表所取用的随机数序号.

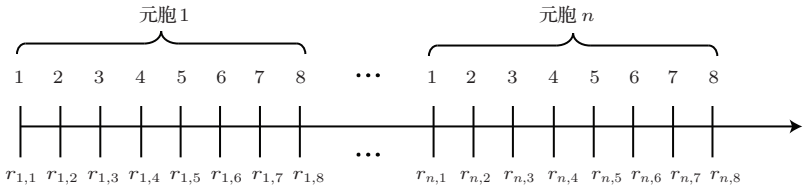


图3 生成并行计算随机数的示意图

因为伪随机数有周期性, 所以设计时应尽量避免不同片段发生重叠. 此外, 为了使不同时间步上同一个线程使用的随机数序列的片段不一样, 每次起始的种子需要引入时间步变量. 具体步骤如下: 首先将线程的全局惟一编号赋值给$n$; 然后从显存中读入当前时间步数据; 再将$n$、时间步数据作为变量输入随机数发生器, 产生两个不同的随机数, 将其存储为随机数状态; 最后进行14个周边元胞的循环计算, 每次需要随机数时, 将上次存储的随机数状态输入随机数发生器, 产生两个新的随机数, 替代旧的随机数状态, 并由这两个新的随机数生成正态分布随机数$N$, 用$N$进行随机力的计算.

经过测试可知, 随着时间步的不断增加, 方法一随机数发生器取用随机数片段的重叠在所难免, 从而可能产生粒子密度的规律性波动, 这种现象需要加以避免.

方法二对应的随机数发生器的特点是不依赖于上一个随机数. 本文采用的是TEA加密算法[26]. 通过给定的密匙, 将两个粒子的编号互相交叉参与比特移位运算、XOR运算、加法运算并循环32次得到随机数. 为了使不同时刻相同粒子对之间相互作用所需要的随机数不一样, 我们将时间参数引入密匙, 从而可以保证所有随机数在空间和时间上都没有依赖性和周期性. 具体步骤如下: 首先从显存中读入当前时间步数据; 再进行14个周边元胞的循环计算, 每次需要随机数时, 将两个相互作用粒子的编号、用户自定义密匙、时间步数据作为输入参数, 通过两个不同参数的TEA随机数发生器产生两个不同的随机数, 并由这两个随机数生成正态分布随机数$N$, 用$N$进行随机力的计算.

方法二随机数发生器的优点在于不依赖于前一个随机数, 完全独立, 从根本上规避了方法一随机数发生器取用随机数片段重叠的风险, 不过方法二算法复杂、计算量较大.

## 4.4 负载平衡

负载平衡是影响并行计算效率的重要因素之一[27]. 以DPD为例, 当每个线程对应一个元胞时, 该线程负责计算该元胞内所有粒子的受力. 如果每个元胞内的粒子数接近, 那么每个线程的计算量均衡, 所有线程将会几乎同时完成计算, 进入下一步. 而当模拟气液两相等流体密度有较大差别的情况时, 每个元胞内的粒子数差距悬殊, 此时负载小的线程将等待负载大的线程, 如图4 所示. 针对负

载差距大的情况, 一种优化算法是在发射 kernel 前将任务排序, 任务量相近的线程分在一个 block, 如图 5 所示. 此方法将大幅度提高 GPU 负载的平衡度, 但是预处理也有相应的成本 (排序耗时和额外的存储空间).

## 4.5 访存优化

由于存储器分为系统内存和显卡内存两部分, CUDA 线程无法直接读取到系统内存上的数据, 所以 CUDA 加速的一项开销就是需要在 device 端重新定义需要使用的变量, 将 host 端的数值拷贝过去, 待计算完成再将数据重新拷贝回来. 由于 host 端与 device 端之间的数据传输依赖于 PCI-E 总线, 传输速率远远低于 CPU 与内存之间或 CUDA core 与显存之间的传输速率, 所以如果算法是需要频繁在 host 端与 device 端之间传输数据, 那传输数据的时间消耗将抵消部分加速计算节省的时间, 甚至使并行程序比串行程序更慢. 所以粒子受力、速度更新、Cell-List 数组更新等工作全部放在 device 端执行, 待计算结束后再拷贝回 host 端输出.

此外, 要使 GPU 加速获得高的加速比, 在 device 端提高线程获取数据的速度也很重要, 包括利用好 GPU 独特的共享存储器、纹理存储器以及对显存的合并访问等.

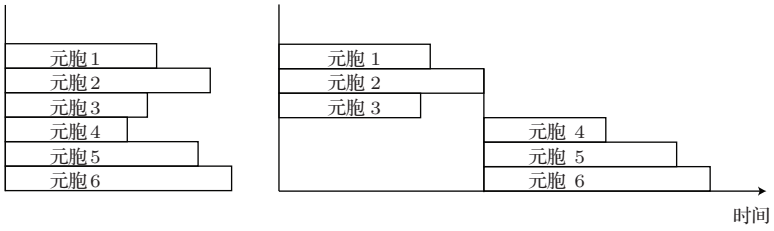


图 4 负载不均衡对并行效率影响的示意图

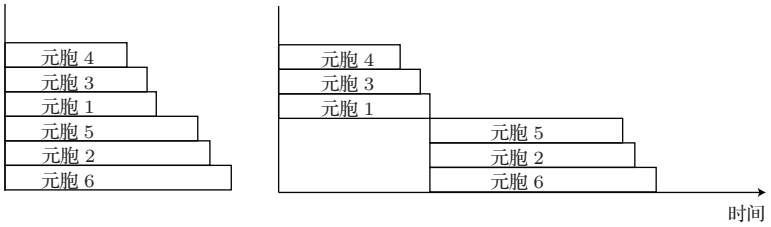


图 5 通过对负载排序预处理优化并行效率的示意图

### 4.5.1 合并访问

在 GPU 内部, 32 根线程组成一个 warp. 一个 warp 内的线程有绝对的同步性, 这是由硬件决定的. warp 内相邻的线程读取显存中相邻位置的数据, 称为合并访问, 这可以由数据结构的调整和算法进行优化. 测试表明, 对相同数组进行相同操作, 合并访问和非合并访问的时间差距可达 32 倍.

### 4.5.2 纹理存储器

纹理存储器跟 GPU 的图形处理有关. 对通用计算而言, 它与常数存储器一样是显存的一部分. 它可以由 CPU 读取和写入, GPU 只能读取. 纹理存储器相对全局存储器的优势在于它有缓存, 可以过滤数据访问、提高访问速度. 相对于常数存储器, 纹理存储器的优势在于其存储空间远大于常数存储器的 64 K, 几乎是显存上剩余的空间. DPD 程序中的一些参数 (如权重、截断半径、密度、温度、重力加速度等) 可以放在纹理存储器中. GPU 的存储架构如图 6 所示.

## 4.6 发射参数的选择

同样的计算域大小, kernel 的发射可以有几十种参数选择, 不同的发射参数会带来不同的性能, 下面主要从 GPU 占用率和寄存器的使用限制来讨论发射参数的选择.

GPU 占用率是指一个流多处理器 (SMX) 上活跃的 warp 数目与最大可驻留 warp 数目的比值. SMX 上的线程无法达到 100% 占用率, 其原因众多. 以计算能力 3.5 的硬件为例: 每个 SMX 最多容纳 64 个 warp; 一个 SMX 最多容纳 2048 个线

程; 一个SMX最多容纳16个block; 一个SMX只有65536个32-bit的寄存器; 一个线程最多控制255个寄存器; 一个SMX最多只有49152 Byte共享存储器等. 发射参数和kernel的设计使得硬件达到以上任何一项的限制就会导致占用率无法提高.

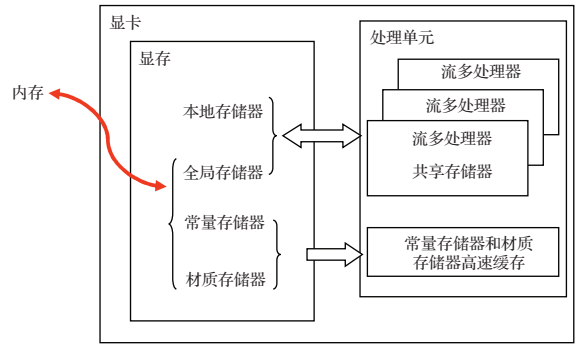


图6 GPU的存储架构

此外寄存器也是影响GPU占用率的因素. 当编写的kernel需要的寄存器很多时, SMX中剩余的线程由于没有足够的寄存器也将排队等待. 我们可以通过编译器限制每个线程的寄存器使用量, 从而让SMX上同时发射尽可能多的block. 然而GPU占用率并不是高效率计算的惟一参考标准. 由于限制了每个线程的寄存器使用量, 不能使用寄存器的局部变量将被分配到本地存储器, 这些变量的读取将变慢. 如何获得最佳的发射参数, 需要权衡多个因素. 通过测试可知, 多数情况下通过限制寄存器使用量提高GPU占用率并不能提高计算效率.

## 5 Poiseuille流动的GPU并行计算

基于上述的GPU并行计算方法, 本文模拟了Poiseuille流动, 并与理论解以及CPU串行计算结果进行了对比. 硬件配置为Intel i5-3450 四核处理器 (3.1 GHz主频)、8GDDR3内存、NVIDIA K20c计算卡 (705 MHz核心频率、4GGDDR5 显存). 计算域如图7所示, 参数设置列于表3.

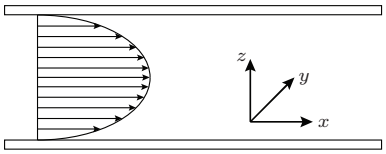


图7 Poiseuille流动计算域示意图

模拟采用Fan等[28]提出的壁面边界条件. 计算区域按$z$轴方向分割成30层, 每一层内每一时间步对粒子速度做一次统计平均, 开始计算的10000步不施加体积力. 10000步后每10000步进行一次统计, 求得该时间段内的平均速度. 图8显示了在130000步时, 采用CPU串行程序、GPU并行程序获得的Poiseuille流动速度与理论结果的对比, 可以看出三者符合得很好.

表3 Poiseuille流动参数设置

| 模拟参数 | 参数值 |
|---|---|
| 截断半径 | 1.0 |
| 保守力系数 | 18.75 |
| 随机力系数 | 3.0 |
| 密度 | 4.0 |
| 时间步长 | 0.02 |
| 体积力 | 0.02 |
| 模拟步数 | 130000 |
| 平衡步 | 10000 |

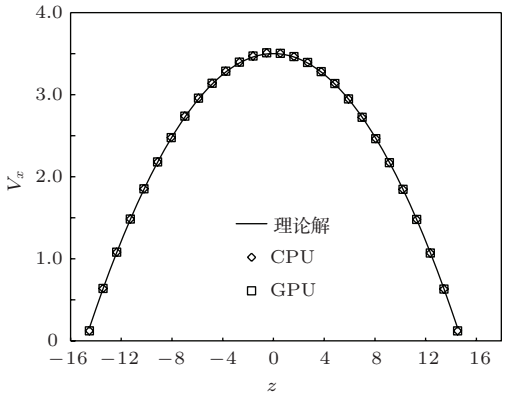


图8 采用CPU串行程序、GPU并行程序获得的Poiseuille流动速度$V_x$与理论结果的对比

在不同的计算域条件下, GPU并行计算每秒粒子更新速度不同, 其对比CPU串行计算粒子更新速度时将具有不同的加速比, 如表4所列. 由表4可知: 粒子数较多时, GPU的计算核心才能全部参加运算; 当每个计算核心上的任务足够多时, 才能掩盖向全局内存读取写入数据的时间延迟.

## 6 突扩突缩通道流动的GPU并行计算

周期性的突扩突缩通道如图9所示, $x$方向总长度为128, 窄通道的长度为48, $y$方向宽度为48,

高度为64, 包含101760个壁面粒子和1130496个流体粒子. 在采用相同参数的情况下, 分别利用GPU和CPU计算突扩突缩通道流动速度, 并在宽通道中央处和窄通道中央处作$x$方向的速度统计, 对比结果如图10 所示. 对于突扩突缩通道流动速度的计算, GPU获得了较CPU21倍的加速比.

表4 计算不同粒子系统时CPU, GPU的粒子更新速度$v_{\rm CPU}$, $v_{\rm GPU}$

| 计算域(粒子数) | $v_{\rm CPU}$/粒子·s$^{-1}$ | 线程映射元胞 | | 线程映射粒子 | |
|---|---|---|---|---|---|
| | | $v_{\rm GPU}$/粒子·s$^{-1}$ | 加速比 | $v_{\rm GPU}$/粒子·s$^{-1}$ | 加速比 |
| $6\times4\times8$ (864) | $1.65\times10^5$ | $5.25\times10^4$ | 0.32 | $1.51\times10^5$ | 0.92 |
| $64\times8\times32$ (67584) | $3.10\times10^5$ | $1.74\times10^6$ | 5.61 | $3.69\times10^6$ | 11.90 |
| $256\times16\times32$ (540672) | $3.11\times10^5$ | $3.16\times10^6$ | 10.16 | $5.23\times10^6$ | 16.82 |
| $256\times32\times32$ (1081344) | $3.07\times10^5$ | $3.59\times10^6$ | 11.69 | $5.82\times10^6$ | 19.15 |

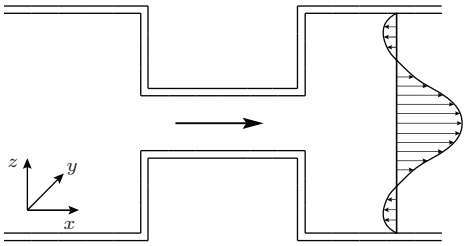


图9 突扩突缩通道流动计算域示意图

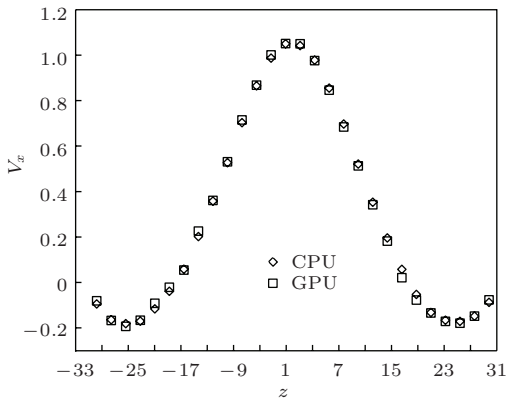


图10 通道截面$x$方向流动速度$V_x$剖面图

## 7 结 论

本文对DPD方法实施了GPU并行加速, 并用Poiseuille流动和突扩突缩通道流动进行了验证计算. 结果表明: 在GPU并行的Cell-List数组更新过程中, 可利用原子操作防止线程冲突; 并行程序采用TEA加密算法替代串行程序的线性随机数发生器, 可以避免并行程序采用线性随机数发生器取用随机数片段重叠的问题; 线程映射粒子的算法比线程映射元胞的算法减少了线程间的等待, 加快了并行计算的粒子更新速度, 且在粒子数较少的系统中, 由于发射了更多的线程, 从而提高了GPU 的利用率, 具有更显著的提速效果; 将权重等常量分配到纹理存储器优化内存访问可以提高计算速度, 并且当系统粒子数量足够多时才能发挥GPU的计算能力.

## 参考文献

[1] Zhang M K, Chen S, Shang Z 2012 *Acta Phys. Sin.* **61** 034701 (in Chinese) [张明焜, 陈硕, 尚智 2012 物理学报 **61** 034701]

[2] Liu H T, Liu M B, Chang J Z, Su T X 2013 *Acta Phys. Sin.* **62** 064705 (in Chinese) [刘汉涛, 刘谋斌, 常建忠, 苏铁熊 2013 物理学报 **62** 064705]

[3] Xu S F, Wang J G 2013 *Acta Phys. Sin.* **62** 124701 (in Chinese) [许少锋, 汪久根 2013 物理学报 **62** 124701]

[4] Chang J Z, Liu M B 2010 *Acta Phys. Sin.* **59** 7556 (in Chinese) [常建忠, 刘谋斌 2010 物理学报 **59** 7556]

[5] Wang X L, Chen S 2010 *Acta Phys. Sin.* **59** 6778 (in Chinese) [王晓亮, 陈硕 2010 物理学报 **59** 6778]

[6] Wu S F, Li X F 2007 *Chin. Phys. Lett.* **25** 184

[7] He L L, Zhang R F, Ji Y Y 2012 *Chin. Phys. B* **21** 088301

[8] Liu C F, Ni Y S 2008 *Chin. Phys. B* **17** 4554

[9] Zhong C W, Xie J F, Zhuo C S, Xiong S W, Yin D C 2009 *Chin. Phys. B* **18** 4083

[10] Li Q D, Tan Y L, Yang F Y 2011 *Acta Phys. Sin.* **60** 030206 (in Chinese) [李清都, 谭宇玲, 杨芳艳 2011 物理学报 **60** 030206]

[11] Li Q D, Zhou H W, Yang X S 2012 *Acta Phys. Sin.* **61** 040503 (in Chinese) [李清都, 周红伟, 杨晓松 2012 物理学报 **61** 040503]

[12] Wang X C, Yan B, Liu H K, Li L, Wei X, Hu G E 2013 *Acta Phys. Sin.* **62** 098702 (in Chinese) [汪先超, 闫镔, 刘宏奎, 李磊, 魏星, 胡国恩 2013 物理学报 **62** 098702]

[13] Huang C S, Zhang W H, Hou Z M, Chen J H, Li M J, He N Z, Shi B C 2011 *Chin. Sci. Bull.* **56** 2829 (in Chinese) [黄昌盛, 张文欢, 侯志敏, 陈俊辉, 李明晶, 何南忠, 施保昌 2011 科学通报 **56** 2829]

[14] Zheng Y C, Liu S, Xiong S W, Zhou J F 2010 *Sci. Tech. Eng.* **7** 1684 (in Chinese) [郑彦奎, 刘沙, 熊生伟, 周季夫 2010 科学技术与工程 **7** 1684]

[15] Zhang D D, Xu Y, Xu L 2012 *Comput. Sci.* **39** 296 (in Chinese) [张丹丹, 徐莹, 徐磊 2012 计算机科学 **39** 296]

[16] Li C G, Maa Jerome P Y, Kang H G 2012 *Sci. China: Phys. Mech. Astron.* **55** 1894

[17] Januszewski M, Kostur M 2010 *Comput. Phys. Commun.* **181** 183

[18] Yuen D A, Wang L 2013 *GPU Solutions to Multi-scale Problems in Science and Engineering* (Berlin: Springer-Verlag) p143

[19] Wu H, Xu J B, Zhang S F, Wen H 2011 *IEIT J. Adapt. Dyn. Comput.* **4** 26

[20] Wang S B, Xua J B, Wen H 2013 *Comput. Phys. Commun.* **184** 2454

[21] Chen S, Jin Y B, Zhang M K, Shang Z 2012 *J. Tongji Univ. (Natural Science)* **40** 137 (in Chinese) [陈硕, 金亚斌, 张明焜, 尚智 2012 同济大学学报 (自然科学版) **40** 137]

[22] Phillips C L, Andersonb J A, Glotzer S C 2011 *J. Comput. Phys.* **230** 7191

[23] Howes L, Thomas D 2007 *Efficient Random Number Generation and Application Using CUDA* (Boston: Addison-Wesley Professional) p370

[24] Nandapalan N, Brent R P, Murray L M, Rendell A 2012 *Parallel Processing and Applied Mathematics* (Berlin: Springer-Verlag) p609

[25] Rose G 2011 *IACR Cryptology ePrint Archive* **2011** 7

[26] Wheeler D J, Needham R M 1995 *Fast Software Encryption* (Berlin: Springer-Verlag) p363

[27] Yao P 2010 *M. S. Dissertation* (Hefei: University of Science and Technology of China) (in Chinese) [姚平 2010 硕士学位论文 (合肥: 中国科学技术大学)]

[28] Fan X J, Nhan P T, Yong N T, Wu X H, Xu D 2003 *Phys. Fluids* **15** 11

# Accelerating dissipative particle dynamics with graphic processing unit*

Lin Chen-Sen[1)] Chen Shuo[1)†] Li Qi-Liang[2)] Yang Zhi-Gang[2)]

1) (*School of Aerospace Engineering and Applied Mechanics, Tongji University, Shanghai 200092, China*)

2) (*Shanghai Automotive Wind Tunnel Center, Tongji University, Shanghai 201804, China*)

( Received 10 December 2013; revised manuscript received 13 January 2014 )

**Abstract**

In this paper, the graphic processing unit (GPU) parallel computing of dissipative particle dynamics (DPD) based on compute unified device architecture is carried out. Some issues involved, such as thread mapping, parallel cell-list array updating, generating pseudo-random number on GPU, memory access optimization and loading balancing are discussed in detail. Furthermore, Poiseuille flow and suddenly contracting and expanding flow are simulated to verify the correctness of GPU parallel computing. The results of GPU parallel computing of DPD show that the speedup ratio is about 20 times compared with central processing unit serial computing.

**Keywords:** dissipative particle dynamics, compute unified device architecture, graphic processing unit, parallel computing

**PACS:** 47.61.–k, 47.11.–j, 87.15.N–, 47.10.–g **DOI:** 10.7498/aps.63.104702

* Project supported by the Fundamental Scientific Research Foundation for the Central Universities of China (Grant No. 125065), the National Natural Science Foundation of China (Grant Nos. 51276130, 10872152), and the Specialized Research Fund for the Doctoral Program of Higher Education of China (Grant No. 20120072110037).

† Corresponding author. E-mail: schen_tju@tongji.edu.cn