

# GPU 并行计算大作业

# 中国科学技术大学

# 格子玻尔兹曼方法求解圆柱绕流问题

姓 名: 汪乘雲

学 号: SA21005050

学院: 工程科学学院

专业: 流体力学

授课老师: 谭立湘

日 期: 2022年5月16日

# 目录

—	绪	论 ·····	2
=	程	序实现方法 · · · · · · · · · · · · · · · · · · ·	2
	2.1	格子玻尔兹曼方法 · · · · · · · · · · · · · · · · · · ·	2
	2.2	基于 CPU 并行 · · · · · · · · · · · · · · · · · ·	3
	2.3	基于 GPU 并行 · · · · · · · · · · · · · · · · · ·	4
Ξ	实	验过程	5
	3.1	问题描述·····	5
	3.2	实验设备	6
	3.3	数据结构设计	7
	3.4	流程计算框图 · · · · · · · · · · · · · · · · · · ·	9
四	结	论与分析 · · · · · · · · · · · · · · · · · · ·	9
	4.1	流场结果对比 · · · · · · · · · · · · · · · · · · ·	9
	4.2	运行时间对比	10
	4.3	参数影响	11
	4	3.1 CPU 并行核数 · · · · · · · · · · · · · · · · · · ·	11
	4.	3.2 GPU 线程块大小 · · · · · · · · · · · · · · · · · · ·	12
五	心	得体会	14

## 第一章 绪论

随着计算机技术的发展和现代流体力学计算方法的不断改进,计算流体力学(CFD)已经成为研究流体力学各种物理现象的重要工具。其中主流的算法包括有限体积法,有限差分法以及有限元法等等。

格子波尔茨曼方法 (Lattice Boltzmann Method, LBM) 作为诞生只有 30 多年的新兴算法,其应用一直是 CFD 领域研究的热点问题之一,其理论和应用研究也都取得了长足的进步。作为一种使用微观模型来模拟流体的宏观行为的介观方法,格子玻尔兹曼方法相比于传统的数值方法,可以比较方便的处理复杂的边界条件,计算效率更高效且具有很好的并行性,能够在大规模并行计算机上计算。由于是基于统计热物理的理论基础,所以在某些传统方法无法胜任的领域,比如微尺度流动与换热、多孔介质、磁流体、生物流体等领域,都可以进行有效模拟。

而随着科技技术的发展,计算流体力学针对的问题也越来越复杂,计算能力的需求也急剧增长,CPU 的发展速度已经难以满足。目前,GPU 凭借其强大的浮点计算能力和高数据吞吐量,已成为高性能计算的首选核心。所以,本文采用 OpenMP 和 CUDA 分别对 LBM 算法进行并行加速。

## 第二章 程序实现方法

### 2.1 格子玻尔兹曼方法

研究流体在宏观层次的连续介质方法所使用的是 NS 方程,而微观层次的统计力学 所采用的方程是玻尔兹曼方程:

$$\frac{\partial f}{\partial t} + c \cdot \nabla f + \mathbf{F} \cdot \nabla_{\xi} f = \Omega(f) \tag{1}$$

其中, f 是流体粒子的概率密度分布函数, F 是系统所受的外力,  $\Omega(f)$  是碰撞算子。

标准的格子玻尔兹曼方法采用的是正方形 (二维) 网格或者正方体 (三维) 网格。以本文所选用的二维 D2Q9 模型为例,其中 0 号速度矢量是零矢量,1-8 号速度矢量指向相邻格点,速度矢量和加权系数表达形式为:

$$\mathbf{c} = c \begin{bmatrix} 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \end{bmatrix}$$
 (2)

$$\omega_i = \begin{cases} 4/9, & i = 0\\ 1/9, & i = 1, 2, 3, 4\\ 1/36, & i = 5, 6, 7, 8 \end{cases}$$
(3)

GPU 并行计算 程序实现方法

其中, c 是格子速度。定义在各个速度空间中的粒子分布函数  $f_i(x,t)$ , 在网格上使用 D2Q9 模型对玻尔兹曼方程进行离散, 就得到格子玻尔兹曼方程 (Lattice Boltzmann Equation, LBE) 如下:

$$f_i(x + e_i \delta t, t + \delta t) - f_i(x, t) = \Omega_i(f), i = 0, 1, \dots, 8$$
 (4)

通常,LBE 的程序结构有两种形式,即碰撞-迁移结构和迁移-碰撞结构,以后者计算流程为例:1) 初始化分布函数

$$f_i(x,0), i = 0, 1, \dots, 8$$
 (5)

2) 执行迁移

$$f_i'(x + e_i\delta t, t + \delta t) = f_i(x, t), i = 0, 1, \dots, 8$$
 (6)

3) 施加边界条件 4) 在时刻  $t + \delta t$  执行碰撞

$$f_i(x + e_i \delta t, t + \delta t) = f'_i(x + e_i \delta t, t + \delta t) + \Omega_i(f), i = 0, 1, \dots, 8$$
 (7)

5) 计算宏观量,包括流体密度、速度和压力等

$$\rho = \sum_{i=0}^{8} f_i, \quad u = \left[ \sum_{i=0}^{8} e_i f_i + \frac{1}{2} f \delta t \right] / \rho, \quad p = \rho c_s^2$$
 (8)

6) 重复步骤 2)~5) 直至满足终止条件。

#### 2.2 基于 CPU 并行

基于 CPU 并行计算的目的是利用多个 CPU 或者多个 CPU 核的协同求解同一个问题,其实质是将一个待解决的问题分解为若干个子问题,然后各个子问题均由各个独立的 CPU 核同时进行计算,这就需要各个 CPU 核在计算的过程中频繁地交换数据。根据数据的通信方式,并行计算系统可以分为共享地址空间系统和消息传递系统。

在共享地址空间系统中,共享内存是指多个核心共享一个内存,内存的地址空间是统一的,从而实现多线程通信、同步、互锁等操作。存储器的分布方式分为集中式和分布式,如果存储器是集中式的,所有的 CPU 对内存的访问速度一样,这种系统称为共享内存多处理器系统,如果存储器的分布式的,CPU 对内存的访问速度只与内存的位置有关,这种系统称为分布式共享内存多处理器系统。目前,基于共享内存模式的主流开发标准是 OpenMP,它是一种基于数据并行的编程模式,即对不同的数据执行相同的代码。

在消息传递系统中,每个计算节点都是一个独立的计算机系统,每个节点的存储器都是单独编码,不能跨节点访问,而当节点间需要数据传递,节点会通过发送包含数据的消息。目前,基于消息传递的主流开发标准是 MPI,它是针对多地址空间进行的多进

GPU 并行计算 程序实现方法

程异步并行模式,其特征是显式同步,显式通信,显式的数据映射和负载分配。MPI 都拥有很高的可移植性和优秀的并行加速能力,但是其编码比较复杂、冗长,开发效率低,可靠性低,一个进程出错就会导致程序崩溃,需要长时间的调试。

总的来说,OpenMP 主要是针对细粒度的循环进行并行,编程简单,适合应用于单台独立的计算机,MPI 主要针对粗粒度级别的并行,可操控性更高,能够更好地发挥硬件性能,适合应用于分布式计算机。

#### 2.3 基于 GPU 并行

在1991年以前,CPU是计算机内唯一的通用处理器,包揽了几乎所有的计算任务,当时的操作系统主要是以命令行的形式进行人机交互。从1991年到2001年,微软公司推出具有图形交互功能的Windows操作系统开始在世界市场上流行,极大刺激了图形硬件的发展。随后,S3 Graphics公司推出全球第一款图形加速器,被视作GPU的雏形。

进入21世纪后,GPU产品的性能突飞猛进。以英伟达为代表 GPU 制造商陆续推出支持可编程图形流水线的 GPU产品,图形流水线是现代 GPU工作的通用模式,它是以三维场景为输入,输出二维的光栅图像到显示器的一种技术。而可编程图形流水线为编程者提供程序接口 (API)来实现自定义算法,这是 GPU 通往通用计算领域的一个开端,后来受电脑游戏和三维建模市场需求的驱动,GPU 的计算性能和内存带宽不断提高。自此开始越来越多的开发者开始关注和研究 GPU 在通用计算领域的应用。早期,编程者需要通过调用标准的图形程序接口,如 OpenGL 或 DirectX 来执行计算、存储、通信等操作,因为涉及较多的底层语言,其编码的难度较大并且代码的可移植性差。为此,英伟达公司提出了 CUDA 架构,专为实现 GPU 高性能并行计算而设计。

CUDA 是一个专门为异构并行计算而设计的架构,包括硬件架构(只限英伟达品牌GPU)和指令集。CUDA 相当于一种高级语言,可以兼容 C、C++、Fortran、Java、Python等高级语言,同时还支持多种应用程序编程接口 (Application Programming Interface, API),包括 OpenGL、DirectX、OpenCL等。为了满足人类对计算机信息处理能力越来越高的要求,英伟达公司已经推出多个系列拥有高性能计算能力的 GPU。如图 1所示,近年推出的英伟达 GPU 在浮点数计算能力上已经遥遥领先于同期的 CPU。随着英伟达 GPU 和CUDA 并行编程模型的不断发展、完善和成熟,CUDA 已经被广泛应用于计算机视觉、人工智能、区块链、计算流体力学等各个领域,并且取得不俗的成绩。

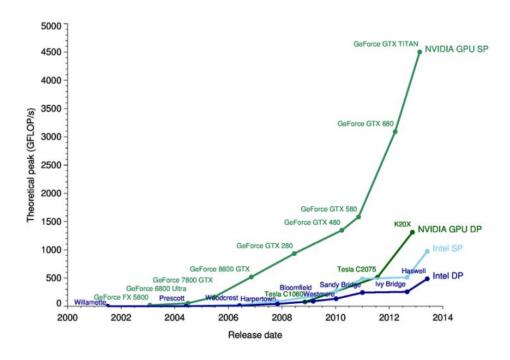


图 1: CPU 和 GPU 单双精度浮点计算能力发展

## 第三章 实验过程

#### 3.1 问题描述

本问题研究的是简单的圆柱绕流问题,物理示意图如图 2所示。

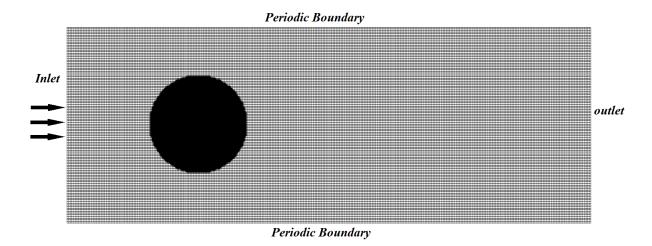


图 2: 计算流场物理示意图

其中计算域的网格  $N_x \times N_y$  划分为 420×160, 圆柱的直径为 80 个网格, 圆心的 x 坐标位于  $0.25N_x$  处, y 坐标位于  $0.5N_y$  处。对于宏观量,我们给定进口来流速度 vxin=0.04,

密度为 roout = 1.0, LBM 松弛时间  $\tau = 0.51$ , 这是与流体粘性有关的参数。

对于边界条件,上下边界条件我们采用周期边界条件,左侧进口边界条件根据给定的人口速度 vxin 设置为 Zou-He 边界条件,右侧出口边界条件采用简单的外推边界条件,圆柱的固壁边界使用 bounceback 边界条件,具体实现代码在函数 apply BCs() 中。

#### 3.2 实验设备

本文实验平台:操作系统为 Microsoft Windows 10。CPU 为 Intel(R) Core(TM) i5-10500 CPU,主频为 3.10 GHz, 六核心十二线程。显卡为 NVIDIA GeForce 1050Ti,显存为 4GB,显卡的核心频率为 1291-1392MHz,部分参数和 GPU-Z 识别结果如图 3和 4所示。

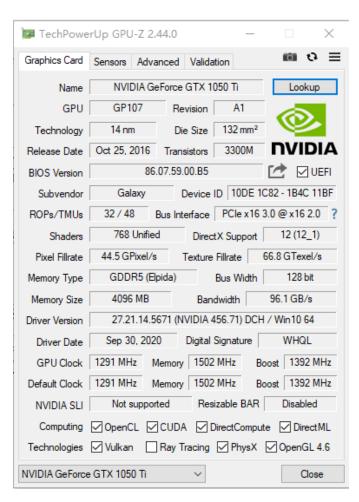


图 3: GPU-Z 设备识别结果

```
Microsoft Visual Studio 调试控制台
  :\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0\1_Utilities\deviceQueryDrv\../../bin/win64/Debug/
CUDA Device Query (Driver API) statically linked version
Detected 1 CUDA Capable device(s)
Device 0: "GeForce GTX 1050 Ti"
CUDA Driver Version:
CUDA Capability Major/Minor version number:
                                                                                    11. 1
                                                                                   6.1
4096 MBytes (4294967296 bytes)
768 CUDA Cores
1392 MHz (1.39 GHz)
   Total amount of global memory:
(6) Multiprocessors, (128) CUDA Cores/MP:
   GPU Max Clock rate:
   Memory Clock rate:
Memory Bus Width:
                                                                                     128-bit
                                                                                   1048576 bytes

1D=(131072) 2D=(131072, 65536) 3D=(16384, 16384, 16384)

1D=(32768), 2048 layers

2D=(32768, 32768), 2048 layers
   L2 Cache Size:
   Max Texture Dimension Sizes
   Maximum Layered 1D Texture Size,
Maximum Layered 2D Texture Size,
                                                            (num) layers
                                                            (num) layers
                                                                                    65536 bytes
49152 bytes
   Total amount of constant memory:
   Total amount of shared memory per block: 49152
Total number of registers available per block: 65536
   Warp size:
                                                                                    2048
   Maximum number of threads per multiprocessor:
   Maximum number of threads per block:
                                                                                    1024
   Max dimension size of a thread block (x,y,z):
Max dimension size of a grid size (x,y,z):
                                                                                   (1024,
                                                                                             1024, 64)
83647, 65535, 65535)
                                                                                  (1024, 1024, 64)
(2147483647, 65535, 65535)
512 bytes
2147483647 bytes
Yes with 5 copy engine(s)
   Texture alignment:
   Maximum memory pitch:
   Concurrent copy and kernel execution:
Run time limit on kernels:
Integrated GPU sharing Host Memory:
                                                                                    Yes
                                                                                    No
   Support host page-locked memory mapping:
                                                                                    Yes
   Concurrent kernel execution:
                                                                                    Yes
   Alignment requirement for Surfaces:
Device has ECC support:
CUDA Device Driver Mode (TCC or WDDM):
Device supports Unified Addressing (UVA):
Device supports Managed Memory:
Device supports Compute Preemption:
Supports Comp
                                                                                   Yes
Disabled
WDDM (Windows Display Driver Model)
                                                                                    Yes
                                                                                    Yes
   Supports Cooperative Kernel Launch:
Supports MultiDevice Co-op Kernel Launch:
Device PCI Domain ID / Bus ID / location ID:
                                                                                    Yes
   Compute Mode:

    Compute Mode:
    Coefault (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

 Result = PASS
```

图 4: deviceQuery 运行结果

#### 3.3 数据结构设计

对于 C 语言的串行和 OpenMP 并行程序,我们分配长度为  $N_x \times N_y$  的十八个一维 float 型数组,用于存储九个分布函数 f0,f1,...,f8 和临时存储的中间数组 tmpf0,tmpf1,...,tmpf8。同时,我们还需要同样长度的两个一维 float 数组 u,v 和一个一维 int 数组 solid,分别用来存储网格点上的两个速度分量以及圆柱固体点的判别标识 (其中 0 代表 固体点,1 代表流体点)。

对于 CUDA 程序,我们在 CPU 端上同样先分配九个一维 float 型数组 f0, f1, ..., f8, 两个一维 float 数组 u, v 和一个一维 int 数组 solid; 在 GPU 端上使用 cudaMallocPitch() 函数分配对应大小的线性数组 f0\_data, f1\_data, ..., f8\_data, 和 u\_data, v\_data, solid\_data cudaMallocPitch() 在分配内存时,为保证数组的每一行的第一个元素的开始地址对齐,每行会多分配一些字节,以保证每行的字节是 256 的倍数 (对齐),这样可以提高读取效

率,示意图如图 5所示。

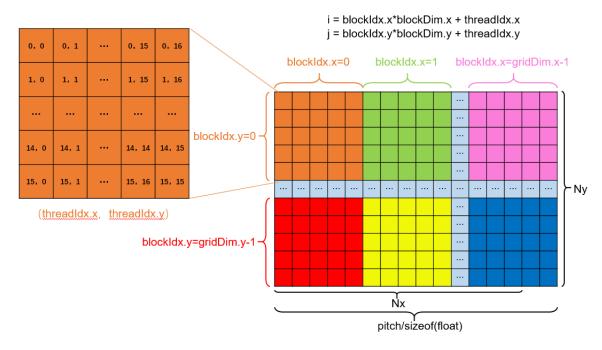


图 5: 流场网格与数组对应逻辑计算结构

因为0方向分布函数f0并不会实际参与对流步,所以我们使用 cudaMallocArray() 仅分配八个 CUDA 数组 f1\_array,f2\_array,...,f8\_array。从 LBM 算法特性可以看出,对流过程每个线程读取位置与邻近线程读取位置非常接近,存在大量的空间局部性,纹理内存非常适用于该算法。我们预先声明八个的 texture<float, 2> 类型的 2D 纹理变量 f1\_tex,f2\_tex,...,f8\_tex,并使用 cudaBindTextureToArray() 将 CUDA 数组与纹理内存绑定起来。

需要注意的是,因为纹理内存作为一种只读内存,无法通过写操作更新,在对流步前后需要不断通过 cudaBindTextureToArray()和 cudaUnbindTexture()进行绑定和解绑操作。从而实现整个流程网格点上分布函数的更新迭代。在核函数 stream\_kernel()中,我们使用 tex2D()来拾取某个网格点周围点的纹理内存,从而实现对流操作。

#### 3.4 流程计算框图

具体的流程计算框图如下:

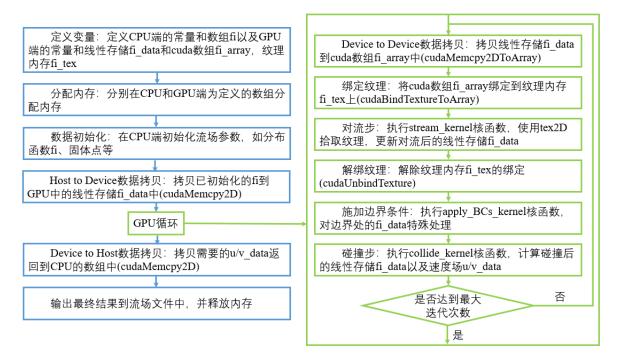
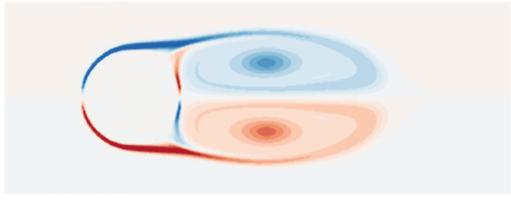


图 6: 流程计算框图

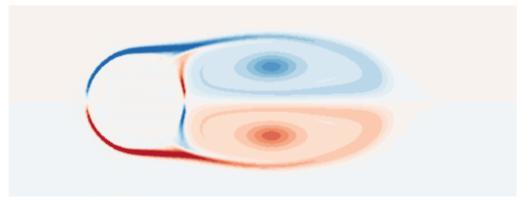
### 第四章 结论与分析

#### 4.1 流场结果对比

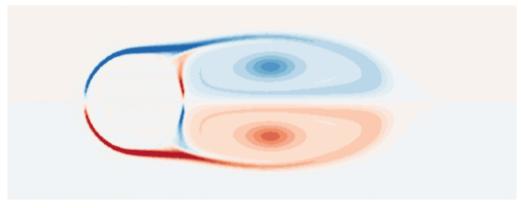
为了验证程序的正确性,我们分别将 LBM\_C.c, LBM\_OpenMP.c 和 LBM\_CUDA.c 三个程序运行 10000 步后的每个网格点的速度分量 u, v 输出到文件 10000.plt 中,并使用后处理软件 tecplot 绘制涡量云图进行对比,涡量计算公式为  $vort = \partial v/\partial x - \partial u/\partial y$ ,可同时验证速度场 u 和 v 的正确性。如图 7所示,三种程序计算结果完全一致,说明三者程序计算正确。



(a) 串行



(b) OpenMP 并行 (四核)



(c) GPU 并行 (block:16 × 16)

图 7: 三种程序计算圆柱绕流的涡量云图

### 4.2 运行时间对比

如表 1所示是我们程序运行时间的结果。这里我们使用两种计时方法:对于串行和 OpenMP 并行程序,我们采用的是 CPU 时钟,通过 clock()调用并且计算。对于 GPU 程序,因为核函数与 CPU 程序是异步执行的,容易带来各种延迟。相比之下,继续采用 CPU 主机端计时就会不够精确 (CPU 计时仅精确到小数点后三位,GPU 时间计时精确

到小数点后六位),因而我们选择采用 CUDA 事件计时 API 计算时间,并且能确定 GPU 不同操作的耗时。

为了减小实验中的随机误差,我们尽量关闭其他后台程序,保证在相同的设备环境下多次运行程序,求出最终消耗时间的平均值以及相对于串行程序的加速比。理想情况下,使用 OpenMP 四核并行最后的加速比应该能达到 4.0,但实际操作过程存在任务切换以及调度等时间开销,最终的加速比也只有 3.4 左右。对于 CUDA 并行,可以发现 GPU 加速比能够达到将近 20,远远优于 OpenMP 的加速效果。考虑到实验所使用的 CPU 和 GPU 价格相近,CPU 想达到相同的加速效果需要的成本大得多,这也是近年来超级计算机越来越多使用 GPU 进行加速计算的原因。

	CPU 串行	OnenMD 皿技	CUDA	
		OpenMP 四核	(block:16×16)	
1	39.960	11.376	2.172047	
2	39.934	11.680	2.183745	
3	39.998	11.407	2.174925	
4	39.984	12.162	2.167604	
5	39.935	11.897	2.182287	
6	39.939	11.712	2.170362	
7	39.925	11.818	2.188568	
8	39.928	12.055	2.175561	
均值	39.950	11.763	2.176887	
加速比	1.000	3.396	18.35206	

表 1: 各程序计算时间对比

#### 4.3 参数影响

#### 4.3.1 CPU 并行核数

众所周知,加速比是衡量一个并行算法是否有效的重要指标,但实际应用中往往还要综合考虑并行效率来选择合适的硬件环境。并行效率,即用加速比除以并行核数,值越高的程序表明越充分发挥了处理器的作用。

为了研究 CPU 核数对并行效率的影响,我们通过 omp\_set\_num\_threads()函数改变并行核数,分别研究了 1/2/4/6 核下的并行计算效率,相关结果在表 2和图 8中。如图所示,随着核数增加,加速比也逐渐增加,但并不是呈线性相关性。在六核时加速比只有 3.822,相比四核提升不大。除了程序中存在串行部分外,还因为即使计算机关闭大部分应用程序,依然有一些系统程序占用 CPU 运行,所以并不能完全发挥六核并行的效果。

	单核	双核	四核	六核
1	39.960	21.372	11.376	10.398
2	39.934	21.473	11.680	10.593
3	39.998	21.410	11.407	10.225
4	39.984	21.259	12.162	10.655
5	39.935	21.223	11.897	10.582
6	39.939	21.335	11.712	10.494
7	39.925	21.219	11.818	9.964
8	39.928	21.295	12.055	10.717
均值	39.950	21.323	11.763	10.454
加速比	1.000	1.874	3.396	3.822
并行效率	1.000	0.93678	0.849042	0.636954

表 2: 不同 CPU 核数并行程序计算时间对比

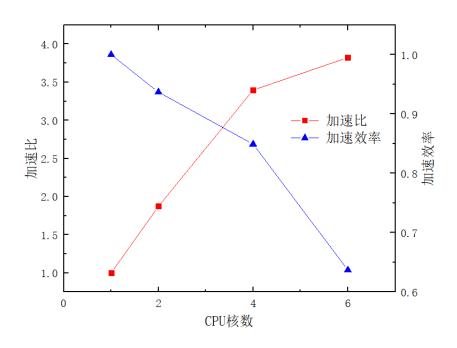


图 8: 不同 CPU 核数并行程序加速比和并行效率对比

#### 4.3.2 GPU 线程块大小

在 CUDA 架构下,执行时的最小单位是 thread,若干个 thread 可以组成一个 block,每一个 block 所能包含的 thread 数目是有限的 (最多 1024 个),执行相同程序的 block 又可以组成 grid。

但从线程运行的原理来看, block 中的线程数量并不是全部使用完最好, 相反在达到一定大小后, 一味地增加线程也不会取得性能的提升, 反而有可能会让性能下降。线

线程块大小	8×8	16×16	16×32	32×16	32×32
1	2.260829	2.172047	2.188429	2.219378	2.238027
2	2.255240	2.183745	2.190795	2.217793	2.233487
3	2.254273	2.174925	2.189309	2.217046	2.235491
4	2.258347	2.167604	2.186552	2.229166	2.245697
5	2.258553	2.182287	2.198995	2.217546	2.236075
6	2.275972	2.170362	2.190464	2.218092	2.242517
7	2.278869	2.188568	2.195127	2.221389	2.244022
8	2.275238	2.175561	2.185740	2.216016	2.234186
均值	2.264665	2.176887	2.190676	2.219553	2.238688
加速比	17.640743	18.352063	18.236558	17.999297	17.845443

表 3: 不同 GPU 线程块大小并行程序计算时间对比

程数达到隐藏各种延迟的程度后,之后线程数量的提升就并无太大的作用。所以,存在一个最优的 block 大小,使得加速效果最好。

这里,我们分别实验了 block 大小为  $8\times 8$ , $16\times 16$ , $16\times 32$ , $32\times 16$ , $32\times 32$  五种情况,并将计算结果统计在表 3和图 9中。从图中可见,线程块的大小过大或者过小都会让加速效果下降,最优的线程块配置在  $16\times 16$  附近。

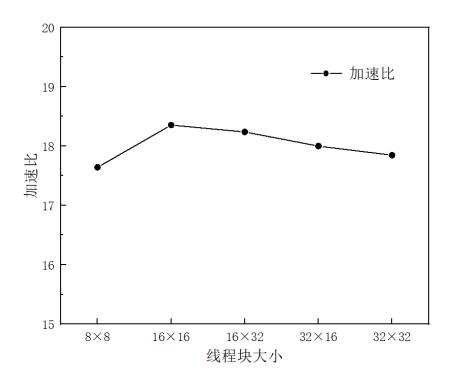


图 9: 不同 GPU 线程块大小并行程序加速比对比

## 第五章 心得体会

通过完成本次 GPU 并行计算的大作业,我回顾了所学的有关 C 语言的相关理论,同时也再次加深对 LBM 算法实现的理解。

首先是对这次实验的一些经验性的总结: (1) 对于 CUDA 的安装,一定要选择适配自己的显卡版本,且对于这种底层的语言安装最好安装在 C 盘,否则在设置环境变量以及调用库函数时,经常会找不到路径; (2) 程序对比实验时,一定要保证属性设置完全一致,Release 版本的程序相比 Debug 版本也有很大的速度优化,除此以外 O1/O2/O3 优化也会对程序速度产生很大影响。(3) 对于线程块的大小,并不是越大越好,存在最优的配置,需要靠自行实验得出。

作为计算流体力学专业的学生,事实上本课题组使用的很多 Fortran 代码都使用了并行计算的方法,包括 MPI 或者 OpenMP 并行。CUDA 作为一种新兴的并行架构,对于加速流体力学计算有很广泛的前景。这次模拟的问题只是流体力学中很简单的问题,实际问题计算中往往还涉及到复杂的流固耦合问题等等,这对于如何使用 CUDA 实现还需要重新设计程序结构。

最后,要感谢老师上课的讲解,虽然之前一直在使用课题组相关并行代码,但对并 行计算的内容了解甚少。同时也要感谢同学的建议,在代码 debug 的方法对我的启发很 大,才能顺利完成本次大作业。